

ANALYSIS-ORIENTED TWO-LEVEL GRAMMARS

by

DAVID ANTHONY WATT

A thesis presented to
the University of Glasgow
in application for the degree of
Doctor of Philosophy

January 1974

SUMMARY

This thesis investigates the twin problems of syntax definition and syntax analysis of programming languages. Conventionally, the syntax of a programming language is defined, partly, by a context-free grammar, leaving non-context-free features of the syntax to be described, say, in English. Correspondingly, the context-free part of the syntax analysis is performed by a parser constructed rigorously from the context-free grammar, while the remaining features are analysed by ad-hoc methods, which tend to be both more difficult to implement and less reliable than the formal methods.

Accordingly, various attempts have been made to devise a formal method of defining completely the syntax of programming languages. Currently, the most promising line of research in this field is the two-level grammar, so called because on one (lower level) context-free grammar is superimposed a second (higher level) grammar. The primary advantage of the two-level grammar lies in its being an extension of the context-free grammar. This makes it possible to extend to two-level grammars some of the many results already obtained for context-free grammars, in particular the techniques of parser construction.

The prototype two-level grammar was van Wijngaarden Form, which was used to define the syntax of ALGOL 68. Although an elegant formalism in many ways, van Wijngaarden Form was not designed to be, and is not, suitable for parser construction.

Affix grammars were devised by Koster with the parsing problem very much in mind. These have a more rigid structure than van Wijngaarden grammars, but their power is enhanced by the addition of arbitrary total recursive functions.

In the first part of this thesis the parsing problem for affix grammars is studied, and a specific parsing technique, based on DeRemer's LR(k) method for context-free grammars, is presented and proved. The class of affix grammars to which this technique is applicable is larger than that to which any other deterministic parsing technique known to the author is applicable.

Affix grammars have the drawback that they are not oriented to human readers. In an attempt to remedy this drawback, and to find a cleaner solution (not involving external functions) to the language definition problem, extended affix grammars are introduced in the second part of this thesis. These are shown to be as powerful as the other two-level grammars. An algorithm for converting any extended affix grammar to an equivalent affix grammar is given.

These results are drawn together to enable the construction of a parser from a suitable extended affix grammar to be completely automated. Various possibilities of tuning the resulting parser are discussed. A specific method of implementation of the parser on a computer is described, and the results of an empirical investigation into the efficiency of the parser, based on this implementation, are presented. The possibility of incorporating the parser into a practical compiler is discussed.

The thesis concludes by indicating possible lines of further development. An extended affix grammar defining the syntax of a complete programming language is presented in the Appendix.

ACKNOWLEDGEMENTS

The work reported in this thesis was started while I was employed by Glasgow University Computing Service and was completed in Glasgow University Computing Science Department on a Science Research Council grant.

I wish to thank my supervisor, Professor D.C. Gilles, for his advice, encouragement, and provision of research facilities. I also wish to acknowledge my debt to Mr John W. Patterson, who provided the intellectual stimulus which motivated this work.

My very special thanks go to my fiancée, Miss Helen Day, whose assistance with the preparation of this thesis and more intangible encouragement helped me through the dark days.

4/1/74

David A. Watt.

CONTENTS

Summary	1
Acknowledgements	3
Contents	4
List of Abbreviations	6
List of Figures	7
List of Theorems	8
Chapter 0. INTRODUCTION	9
0.1. Two-Level Grammars	9
0.2. Affix Grammars and Extended Affix Grammars	14
0.3. Terminology and Notation	16
Chapter 1. AFFIX GRAMMARS	18
1.0. Introduction to Affix Grammars	18
1.1. Definition of an Affix Grammar	21
1.2. Well-formed Affix Grammars	26
1.3. Formal Properties of Affix Grammars	30
1.4. Some Comments on Affix Grammars	34
Chapter 2. LEFT-TO-RIGHT PARSERS FOR WELL-FORMED AFFIX GRAMMARS	36
2.1. LR(k) Parsing for Context-Free Grammars	36
2.2. Canonical Forms of an Affix Grammar	42
2.3. Auxiliary Grammar of a Well-Formed Affix Grammar	44
2.4. Optimisation to an Auxiliary Grammar	51
2.5. Head Grammar of an Auxiliary Grammar; AF-LR(k) Grammars	55
2.6. Parsers for AF-LR(0) Grammars	60
2.7. Parsers for AF-LR(k) Grammars	67
2.8. Left Recursion in Affix Grammars	68
2.9. A-LR(k) Grammars; Multi-Predicate States	71
2.10. Summary: Construction of an AF-LR(k) Parser	74

Chapter 3.	EXTENDED AFFIX GRAMMARS	77
3.0.	Introduction to Extended Affix Grammars	77
3.1.	Definition of an Extended Affix Grammar	79
3.2.	Well-formed Extended Affix Grammars	81
3.3.	Formal Properties of Extended Affix Grammars	82
3.4.	Conversion of an Extended Affix Grammar into an Equivalent Affix Grammar	85
3.5.	Some Comments on Extended Affix Grammars	93
Chapter 4.	LEFT-TO-RIGHT PARSERS FOR WELL-FORMED EXTENDED AFFIX GRAMMARS	95
4.0.	Introduction	95
4.1.	Parse-time Representation of Affixes	96
4.2.	Implementation of Synthesise-, Analyse- and Equal-Predicate Functions	98
4.3.	Left Recursion in Extended Affix Grammars	102
4.4.	Multi-Predicate States in the Parser	104
4.5.	A Practical Implementation of the Parser	105
4.6.	Optimisations to the Parser Implementation	109
4.7.	Efficiency of the Parser Implementation	111
4.8.	Incorporation of the Parser into a Translator	114
Chapter 5.	CONCLUSIONS	117
5.1.	Applications of Extended Affix Grammars	117
5.2.	Notation	118
5.3.	Further Research	119
Appendix A.	AN EXTENDED AFFIX GRAMMAR FOR PASCAL	122
A.1.	The Definition of PASCAL	122
A.2.	Syntax Analysis of PASCAL based on the EAG	124
List of References		126
Index of Definitions		128

LIST OF ABBREVIATIONS

AG	affix grammar
A-LR(k)	affix-LR(k)
AF-LALR(k)	affix-free-LALR(k)
AF-LR(k)	affix-free-LR(k)
AF-SLR(k)	affix-free-SLR(k)
AFSM	affix finite-state machine
CF	context-free
CFG	context-free grammar
CFSM	characteristic finite-state machine
EAG	extended affix grammar
VWF	van Wijngaarden form
VWG	van Wijngaarden grammar

LIST OF FIGURES

	Near page		Near page
Figure 1.1	24	Figure 4.1	97
Figure 1.2	24	Figure 4.2	100
Figure 1.3	24	Figure 4.3	100
		Figure 4.4	100
Figure 2.1	41	Figure 4.5	100
Figure 2.2	41	Figure 4.6	101
Figure 2.3	41	Figure 4.7	104
Figure 2.4	47	Figure 4.8	107
Figure 2.5	47	Figure 4.9	107
Figure 2.6	54	Figure 4.10	111
Figure 2.7	54	Figure 4.11	113
Figure 2.8	58	Figure 4.12	113
Figure 2.9	65	Figure 4.13	113
Figure 2.10	65	Figure 4.14	113
Figure 2.11	65	Figure 4.15	113
Figure 2.12	73	Figure 4.16	113
Figure 2.13	73		
		Figure 5.1	117
Figure 3.1	80	Figure 5.2	117
Figure 3.2	80	Figure 5.3	118
Figure 3.3	93		
		Figure A.1	125

LIST OF THEOREMS

	Page
Theorem 1.1	29
Theorem 1.2	30
Corollary 1.3	33
Theorem 2.1	43
Lemma 2.2	47
Lemma 2.3	50
Theorem 2.4	51
Theorem 2.5	57
Theorem 2.6	58
Theorem 2.7	66
Theorem 3.1	82
Theorem 3.2	83
Corollary 3.3	85
Theorem 3.4	87
Theorem 3.5	91

CHAPTER 0

INTRODUCTION

0.1. Two-Level Grammars

Ever since the syntax of ALGOL 60 (Naur 60, 63) was defined by a context-free grammar (CFG), CFGs have attracted a considerable amount of attention. A number of other programming languages have since been defined by CFGs. Each grammar serves as a means of communication between the language designer and the programmer, who requires a readable description of the allowable constructs of the language; and between the designer and the implementor, who requires a detailed, accurate and complete specification of the language's syntactic structure for his translator. Practice has shown that a well-designed CFG can simultaneously satisfy almost all these requirements.

The usefulness of CFGs as an aid to the implementation of programming languages has been much enhanced by the development of a variety of syntax-directed parsing techniques (Feldman 68). These techniques enable parsers to be constructed automatically from suitable CFGs. The constructed parsers, in many cases, are almost as efficient as those written by hand, and have linear performance characteristics. They can therefore be incorporated into practical translators. Moreover, their correctness can be guaranteed.

Unfortunately, typical programming languages are not strictly context-free: they are context-sensitive. Examples of features of a typical language which defy description by CFGs are the correspondence between declarations and applications of

identifiers, and the compatibility of actual and formal parameters. In a programming language with generalised data types, even type compatibility cannot be defined by a CFG.

At this point we should state exactly what we mean by "syntax", as there is no general agreement on the definition of this term. One view is that syntax encompasses precisely those features of a language which can be defined by a CFG; context-sensitive features are considered as belonging to the realm of semantics. Apart from making a virtue of necessity, this view seems to us, if it were adopted universally, to tend to inhibit the search for more powerful methods of defining languages. Our own view is that any criterion for well-formedness of a program which can be determined algorithmically is syntactic.

Conventionally, in language definitions the CFG defining the context-free part of the language's syntax is accompanied by descriptions, in a natural language such as English, of the context-sensitive features of the syntax. This is not fully satisfactory, because of the consequent increased dangers of ambiguity, incompleteness, and misunderstanding on the part of the reader. Furthermore, there is no possibility of automatically implementing syntax features described in this way.

The implementor's traditional solution to this problem reflects the context-free view of what constitutes syntax. A context-free parser is constructed from the CFG, and context-sensitive features are transcribed as semantic constraints. In practice this approach has been justified by the use of certain ad-hoc techniques, such as hashed identifier tables, which enable the context-sensitivities to be checked without seriously degrading the performance of the underlying context-free parser. On the other hand, it increases the burden on the post-syntactic phase of the translator, already the most difficult part to implement. Moreover, it is not easy to prove the correctness of such ad-hoc techniques.

Context-sensitive grammars, otherwise known as Chomsky type 1 grammars (Hopcroft 69), are not likely to provide a solution. A context-sensitive grammar for a typical programming language is likely to be long and unreadable. No method of constructing reasonably efficient parsers from a useful class of context-sensitive grammars is known to us.

The useful properties of CFGs suggest that the best approach to this problem is to devise an extension to CFGs which will enhance their power but retain their desirable properties. Early proposals along these lines tended, however, to be designed to handle specific features of programming languages: for example, the formalism used in the definition of the syntax of ALGOL W (Wirth 66) was designed to specify type compatibility, and "property grammars" (Stearns 69) were designed to formalise identification.

Currently the most promising development is the "two-level grammar", so called because on one (lower level) context-free grammar is superimposed a second (higher level) grammar. Interest in this concept dates from the use of one form of two-level grammar to define the syntax of ALGOL 68 (van Wijngaarden 68). Grammars of this form subsequently became known as "van Wijngaarden grammars" (VVGs), and were formalised by Baker (Baker 72).

In a VVG the upper level consists of a set of context-free rules, in which the nonterminals are known as "metanotions". An example of a metanotion (all our examples in this section are taken from (van Wijngaarden 68)) is 'MODE', whose terminal productions include 'real' and 'reference to real'. The lower level contains meta-rules each of which has one "hypernotation" on its left side and a sequence of "hypernotations" on its right side. A hypernotation is written as a string, in which metanotions may be embedded. An example of a hypernotation is 'MODE source'. An example of a VVG meta-rule is

reference to MODE assignation : reference to MODE destination ,
becomes symbol , MODE source .

(The colon separates the left and right sides, and the commas separate adjacent hypernotations.) Such a meta-rule acts as a generator for context-free-like production rules: to generate a production rule, each metanotion is replaced, throughout the meta-rule, by some terminal production of itself. In our example, if we choose to replace 'MODE' by 'real', we obtain the production rule

reference to real assignation : reference to real destination ,
becomes symbol , real source .

In a production rule the hypernotations have been replaced by "protonotions" (such as 'real source'), which have no embedded metanotions. Protonotions play a similar role to nonterminals in a CFG, that is, at each step of a derivation a protonotion may be replaced by the sequence of protonotions on the right side of a production rule if the first protonotion is the left side of that rule.

The requirement that all occurrence of a given metanotion in a meta-rule must be replaced by the same terminal production allows many context-sensitivities to be defined; our example meta-rule specifies that the "destination" (left hand side) and "source" (right hand side) of an "assignation" must have compatible modes, that is, their modes must be the same but for the prefix 'reference to' of the former. Moreover, it is possible to generate an infinite number of production rules from a finite set of meta-rules, since a metanotion may have an infinite number of terminal productions. These properties make VWGs a highly powerful formalism for defining languages, as powerful in fact (Sintzoff 67) as Chomsky type 0 grammars (Hopcroft 69).

The principal disadvantage of VWGs is their very generality. Consider, for example, the meta-rules

- (1) formal LOWER bound : strict LOWER bound option ,
flexible symbol option .
- (2) NOTION option : NOTION .
- (3) NOTION option : .

The metanotion 'LOWPER' has as terminal productions 'lower' and 'upper', and the set of terminal productions of 'NOTION' includes every protonotion. (Recall that a protonotion is written as a string.) Thus, replacing 'LOWPER' in meta-rule (1) by 'lower', we obtain the production rule

formal lower bound : strict lower bound option ,
flexible symbol option .

Similarly replacing 'NOTION' in meta-rule (2) by 'strict lower bound' and by 'flexible symbol' respectively, we obtain the production rules

strict lower bound option : strict lower bound .
flexible symbol option : flexible symbol .

These three production rules might well be applied one after another in a derivation. Our point is that the last two production rules, although generated from the same meta-rule, are not related to each other in any genuine logical or syntactic sense. The existence of meta-rules such as (2) and (3) implies that protonotions (and hypernotations) are somewhat polymorphous objects.

Another feature of VWGs is that protonotions may be infinitely long, since a metanotion may be replaced by an infinite string, and that an infinitely long protonotion may occur in the derivation of a finite program. Consider an example from ALGOL 68. In the reach of the mode-declaration

mode lisp = struct (int h , ref lisp t)

'lisp' is a 'structured with integral field letter h and reference to structured with integral field letter h and reference to field letter t field letter t declarer'.

These observations lead us to conclude that VWGs are not suitable for constructing parsers. We feel intuitively, from the point of view of parsing, that protonotions should have well-defined structures, reflecting the attributes of strings derived from them, and that production rules generated from the same meta-rule should be similar to each other in some syntactic sense.

0.2. Affix Grammars and Extended Affix Grammars

This thesis is a study of two other classes of two-level grammars, namely "affix grammars" and "extended affix grammars", which retain the basic ideas of VWGs and are formally as powerful, but which satisfy the conditions which we consider desirable. The basic ideas which are retained are the upper-level grammar, the embedding of metanotions in hypernotations, and the generation of production rules from meta-rules by systematic replacement of metanotions by terminal productions of themselves.

Affix grammars were devised by Koster (Koster 70) with the parsing problem very much in mind. In affix grammars hypernotations are tightly restricted in form, and they depend for their power largely on the inclusion of arbitrary total recursive functions, which may be invoked from within the rules, and which are defined separately. In chapter 1 of this thesis we present a definition of affix grammars and some of Koster's results for them, and we comment on the merits and drawbacks of these grammars.

In chapter 2 we develop our own approach to the parsing problem for affix grammars. We first of all reduce this problem to a manageable extension of the parsing problem for context-free grammars, to which any one of several well-known techniques could be applied. We choose the LR(k) parsing method as refined by DeRemer (DeRemer 69,71). This generalised LR(k) parsing method is, we believe, applicable to a larger class of affix grammars than any other deterministic parsing technique for affix grammars.

The main drawback of affix grammars is that they tend to be tedious to write and difficult to read. In an attempt to combine most of the desirable properties of VWGs and affix grammars, we propose in chapter 3 a new class of two-level grammars, which we call the "extended affix grammars". Extended affix grammars have the clean appearance of VWGs, and do not invoke external functions. We show, however, that every extended affix grammar can be converted automatically into an equivalent affix grammar, so the

suitability of affix grammars for parser construction is retained.

In chapter 4 we show how to apply our results of chapter 2 to construct parsers from extended affix grammars. The main problem with affix grammars, namely that of implementing the separately defined functions, does not exist with extended affix grammars: all the information required by the constructor is available in the rules of the upper and lower levels. Thus we are able to automate completely the construction of deterministic parsers from suitable extended affix grammars.

We have implemented such a parser on a computer and measured its performance. As expected, it was slower than a syntax analyser for the same language based on a context-free parser, but the difference was small enough, we believe, to justify optimism in the possible development of our parsing method for use in practical translators.

We conclude our thesis in chapter 5 by outlining applications of extended affix grammars and discussing the question of notation, and by setting out possible lines of future research. An appendix contains a major example of the use of an extended affix grammar to define a complete programming language.

0.3. Terminology and Notation

The starting point for the research reported in this thesis is the great volume of results, obtained in a comparatively short period, for formal languages in general and context-free grammars in particular. It is not possible explicitly to acknowledge all this work, only those sources which have been most immediately useful to our research. Most of these sources, of course, were themselves based on earlier work.

Even in these well-developed fields, terminology has not yet become completely uniform, so we have adopted in this thesis the terminology used in our most useful sources. Terminology, definitions and results relevant to context-free grammars and parsing will be found in (DeRemer 69), and those relevant to formal languages in general will be found in (Hopcroft 69).

The study of two-level grammars is comparatively new, and the terminology is correspondingly diverse. As we require a uniform terminology for purposes of comparative evaluation, we have chosen the terminology used in (Koster 70), except for a few terms which, for reasons of clarity or brevity, we have borrowed from (van Wijngaarden 68) or coined ourselves. Terms connected with parsing in two-level grammars are, where appropriate, borrowed from context-free parsing terminology.

We require on occasion a notation for specifying functions, and for this purpose we use λ -expressions. A function with bound variables x_1, \dots, x_n and body E is written as

$$\lambda x_1 \dots \lambda x_n (E) ,$$

where the parentheses may be omitted if the body is already enclosed between parentheses.

If P is a predicate and E_1 and E_2 are expressions, then we write

$$(P|E_1|E_2)$$

to denote an expression whose value is the value of E_1 if the value of P is true, or the value of E_2 if the value of P is false.

We use the notation $[m, n]$ as an abbreviation for the set $\{k \mid m \leq k \leq n\}$.

Finally, the notation

$\exists x : P$

is to be read as "there exists x such that P ", where P is a predicate. Likewise, the notation

$\nexists x : P$

is to be read as "there does not exist x such that P ";

$\exists x_1, \dots, x_n : P$

is to be read as "there exist x_1, \dots, x_n such that P "; etc.

CHAPTER 1

AFFIX GRAMMARS

1.0. Introduction to Affix Grammars

With the idea of devising a grammar system more oriented to parsing than van Wijngaarden grammars (VWGs), Koster defined a new class of two-level grammars, which he called the "affix grammars". Many of the concepts of VWGs, which we explained in section 0.1, have their counterparts in affix grammars. We shall illustrate this introduction to affix grammars by the same example which we used in section 0.1 to illustrate VWGs, namely the definition of legal "assignments" in ALGOL 68.

In an affix grammar, a protonotion is a structured object, consisting of a "head" and a set of "affixes", whose number and domains are fixed by the head. The head characterises a set of protonotions which have broadly similar syntactic properties. Each affix of a protonotion represents some attribute of a phrase which is a terminal production of that protonotion. Thus we might have a head 'source' which characterises the set of right hand sides of ALGOL 68 assignments. Each protonotion whose head is 'source' might have a single affix which represents the mode of a particular source, for example 'real'. The structure of a protonotion is emphasised by writing it with the affix(es) enclosed in parentheses and following the head, for example 'source(real)'.

The domain of each affix is the set of finite terminal productions of some "affix-nonterminal", defined by a set of

context-free "affix-rules", which form the upper level of the affix grammar. (An affix-nonterminal is the same as a metanotion in VWG terminology.) Thus we might have an affix-nonterminal 'MODE' whose terminal productions include 'real', 'reference to real' and 'boolean'.

A hypernotation has the same form as a protonotion, except that an affix-nonterminal may stand in place of each affix; thus 'source(MODE)' is a hypernotation. The meta-rule of our first example of section 0.1 might be written in an affix grammar as

```

assignment(LMODE) : destination(LMODE)
                    check-ref(LMODE, MODE) becomes-symbol
                    source(MODE)

```

where 'LMODE' has the same set of terminal productions as 'MODE'.

As in VWGs, a production rule is generated from a meta-rule by replacing each affix-nonterminal, throughout the meta-rule, by some terminal production of itself. Thus, replacing 'LMODE' by 'reference to real' and 'MODE' by 'real', we obtain the production rule

```

assignment(reference to real) : destination(reference to real)
                               check-ref(reference to real, real)
                               becomes-symbol source(real)

```

Each affix-position in a hypernotation may be occupied only by an affix-nonterminal or by an affix. As this is rather restrictive, the power of affix grammars is enhanced by the addition of total recursive functions over the affixes. In our example, 'check-ref' would invoke a predicate which checks that its first parameter (affix) has the prefix 'reference to' but is otherwise identical to its second parameter. Thus, although by replacing 'LMODE' by 'boolean' in our meta-rule we could generate the production rule

```

assignment(boolean) : destination(boolean)
                     check-ref(boolean, real) becomes-symbol
                     source(real)

```

this production rule could not be used in a derivation, since the predicate invoked by 'check-ref' would yield the result false when

its parameters are the affixes 'boolean' and 'real'.

It must be emphasised that although in this example a fairly straightforward transliteration from a VWG meta-rule to an affix grammar meta-rule was possible, the nature of VWGs makes such a transliteration impossible in general. For example, no affix grammar defining ALGOL 68 would be likely to contain anything similar to the meta-rules

NOTION option : ; NOTION .

which we quoted in our second example of section 0.1.

Koster also defined a subclass of affix grammars, which he called "well-formed affix grammars". A clear distinction is made between "derived" and "inherited" affixes of a protonotion. Given a protonotion and a phrase which is a terminal production of that protonotion, a derived affix of the protonotion, roughly speaking, represents an attribute of the phrase which is determined by the phrase itself, and an inherited affix an attribute determined by the context of the phrase. In addition various restrictions are placed upon the meta-rules, whose effect is to make well-formed affix grammars amenable to syntax-directed parsing by techniques which are extensions of well-known techniques already developed for CFGs.

In this chapter we present a formal definition of affix grammars (section 1.1), a definition of well-formed affix grammars (section 1.2), and an investigation into their formal properties (section 1.3). This material is heavily based upon the work of Koster as reported in (Koster 70), but includes slight modifications to his definitions, notation and terminology, which we attempt to justify. (In our example in this introduction, however, we adhered to Koster's definition in order to simplify the explanation.) Finally, in section 1.4, we comment on the advantages and disadvantages of affix grammars from several points of view.

1.1. Definition of an Affix Grammar

The original definition of an affix grammar was given by Koster (Koster 70). For our own purposes we find it necessary to make certain modifications to this definition. These modifications are discussed later, and it is shown that they make no difference to the power of affix grammars. We also take the liberty of changing his notation in favour of a notation which seems more natural.

We define an affix grammar (AG) to be an 11-tuple

$$G = (V_n, V_t, A_n, A_t, Q, e, R, B, D, S, P)$$

whose elements are defined in the following paragraphs.

V_n is a finite non-empty set of nonterminal symbols, V_t a finite non-empty set of terminal symbols, A_n a finite set of affix-nonterminal symbols, A_t a finite set of affix-terminal symbols, and Q a finite set of primitive predicate symbols. V_n , V_t and Q are mutually disjoint, as are A_n and A_t .

e , a member of V_n , is the distinguished nonterminal.

R is a finite set of affix-rules. Each affix-rule is of the form

$$a : a_1 \dots a_m,$$

where $a \in A_n$, $m \geq 0$, and $a_1, \dots, a_m \in A_n \cup A_t$. Thus, if a is an affix-nonterminal, the 4-tuple $G_a = (A_t, A_n, a, R)$ is a CFG. We denote by $L(a)$ the language generated by G_a , and by L the union of all sets $L(a)$ such that $a \in A_n$. Each string in L is known as an affix. Note that every affix is finite in length.

B is a finite set of affix-variables (or, simply, variables), disjoint from A_t . D is a map from B into A_n : $D(b)$ is the associated affix-nonterminal of the variable b , and $L(D(b))$ is the domain of b , that is the set of affixes which can be derived, using the affix-rules in R , from its associated affix-nonterminal.

S is the control of the AG, a set of 5-tuples

$$S_x = (x, N_x, \tau_x, \alpha_x, F_x),$$

one for each nonterminal and primitive predicate symbol x . N_x is the number of affix-positions of x . τ_x is an N_x -tuple over $\{1, \delta\}$ specifying the types of the affix-positions of x : $\tau_{x,i} = 1$ (or δ) denotes that the i -th affix-position of x is inherited (respectively, derived). α_x is an N_x -tuple over A_n specifying the domains of the affix-positions of x : $L(\alpha_{x,i})$ is the domain of the i -th affix of x . F_x is the associated function of x , and is relevant only if x is a primitive predicate symbol; it is a total recursive function

$$F_x : L(\alpha_{x,1}) \times \dots \times L(\alpha_{x,N_x}) \rightarrow \{\underline{\text{true}}, \underline{\text{false}}\}.$$

P is a finite set of meta-rules. Each meta-rule is of the form

$$Z : Z_1 \dots Z_m,$$

where $m \geq 0$. Z is the left side, and $Z_1 \dots Z_m$ the right side, of this meta-rule. Z is of the form $v(b_1, \dots, b_{N_v})$, where $v \in V_n$ and, for each $i \in [1, N_v]$, $D(b_i) = \alpha_{v,i}$, that is, each affix-position on the left side is occupied by an affix-variable whose associated affix-nonterminal specifies the domain of that affix-position. For each $j \in [1, m]$, Z_j is either a terminal or a hypernotation. A hypernotation is of the form $x(f_1, \dots, f_{N_x})$, where the head $x \in V_n \cup Q$ and, for each $i \in [1, N_x]$, f_i is either an affix-variable or an affix. (Thus the left side of each meta-rule is occupied by a hypernotation of a restricted form.)

A meta-rule is a rule for generating (CF-like) production rules, which may in turn be used for deriving sentences in the AG. Suppose that b_1, \dots, b_n are all the variables occurring in the meta-rule

$$Z : Z_1 \dots Z_m,$$

and that c_1, \dots, c_n are affixes in the respective domains of these variables. Suppose further that, if b_1 is replaced by c_1, \dots , and b_n is replaced by c_n , throughout the meta-rule, the resulting

rule is

$$Y : Y_1 \dots Y_m .$$

Then $Y \rightarrow Y_1 \dots Y_m$ is a production rule, and $Y_1 \dots Y_m$ is a direct production of Y . A protonotion is a hypernotation in which each affix-variable has been replaced by an affix in the domain of that variable. Y will be a protonotion, and each Y_j will be either a terminal or a protonotion (according as Z_j was a terminal or a hypernotation).

A protonotion whose head is a primitive predicate symbol q may or may not have a direct production – the empty string (λ) – depending on the value of q 's associated function when applied to the affixes of the protonotion. We define that $q(c_1, \dots, c_{N_q}) \rightarrow \lambda$ if and only if $F_q(c_1, \dots, c_{N_q})$ evaluates to true.

Without loss of generality, we can assume that the distinguished nonterminal e has no affix-positions and does not occur in the right side of any meta-rule.

A notion is a protonotion which has at least one direct production. In particular, e is a notion. A protonotion which is not a notion is termed a blind alley.

A production of a notion X is either (i) a direct production of X , or (ii) a string of protonotions and terminals obtained by replacing, in a production of X , some notion Y by a direct production of Y . A terminal production of a notion is one which consists entirely of terminals.

A sentence is a terminal production of the distinguished nonterminal e .

The language generated by the AG is the set of all sentences of the AG.

We now give some further definitions which we shall find useful later. All of these are generalisations of terms well known from CFG theory. Consider the AG G defined above; let N denote the set of protonotions of G .

If $Y \in N$, and $\pi, \sigma, \tau \in (N \cup V_t)^*$, then $\sigma Y \tau \Rightarrow \sigma \pi \tau$ if and only if $Y \rightarrow \pi$. We call the relation $\sigma Y \tau \Rightarrow \sigma \pi \tau$ a direct derivation. If $\beta_0, \beta_1, \dots, \beta_n \in (N \cup V_t)^*$ and $\beta_0 \Rightarrow \beta_1 \Rightarrow \dots \Rightarrow \beta_n$ (where $n \geq 0$), then we write $\beta_0 \Rightarrow^* \beta_n$ and call this a derivation from β_0 to β_n .

The language generated by G can alternatively be defined by

$$\mathcal{L}(G) = \{ \tau \in V_t^* \mid e \Rightarrow^* \tau \} .$$

In general there is more than one derivation from e to a given sentence, since the protonotions occurring in an intermediate string can be replaced in any order by direct productions. We choose as our canonical derivation the right derivation, in which the rightmost protonotion in each intermediate string is replaced by a direct production. Formally, $\sigma Y \tau \Rightarrow \sigma \pi \tau$ is a canonical direct derivation if and only if $Y \rightarrow \pi$ and $\tau \in V_t^*$. A canonical derivation is a derivation of which every step is a canonical direct derivation. As we are interested only in canonical derivations, we shall henceforth use the relations \Rightarrow and \Rightarrow^* only in this sense.

A canonical form is a string of protonotions and terminals which can be canonically derived from e .

A canonical parse (or, simply, a parse) of a canonical form β is the reverse of the sequence of production rules applied in a canonical derivation from e to β .

An AG is unambiguous if and only if every canonical form has a unique canonical parse.

$V_n = \{ \text{block, declns, stmts, stmt, vble, tag, type} \}$

$V_t = \{ \text{var, begin, :, :=, x, y, z, int, bool, end} \}$

$A_n = \{ \text{TAG, MODE, LIST} \}$

$A_t = \{ \text{x, y, z, i, b} \}$

$Q = \{ \text{empty, declare, equal, identify, tagx, tagy, tagz, modei, modeb} \}$

$e = \text{block}$

R:-

TAG : x ; y ; z .
 MODE : i ; b .
 LIST : ; LIST TAG MODE .

$B = \{ T, M, M1, L, L1 \}$

$D = \{ (T, \text{TAG}), (M, \text{MODE}), (M1, \text{MODE}), (L, \text{LIST}), (L1, \text{LIST}) \}$

$S = \{$
 (block, 0, -, -, -), (declns, 1, δ , LIST, -),
 (stmts, 1, ι , LIST, -), (stmt, 1, ι , LIST, -),
 (vble, 2, (ι, δ), (LIST, MODE), -),
 (tag, 1, δ , TAG, -), (type, 1, δ , MODE, -),
 (empty, 1, δ , LIST, $\lambda l (l=\lambda)$),
 (declare, 4, ($\iota, \iota, \iota, \delta$), (LIST, TAG, MODE, LIST),
 $\lambda l \lambda t \lambda m \lambda k ((\exists p, q : l=ptq) \wedge k=l\text{tm})$),
 (equal, 2, (ι, ι), (MODE, MODE), $\lambda m \lambda n (m=n)$),
 (identify, 3, (ι, ι, δ), (LIST, TAG, MODE),
 $\lambda l \lambda t \lambda m ((\exists p, q : l=ptmq) \wedge (\exists r, s : p=rts))$),
 (tagx, 1, δ , TAG, $\lambda t (t=x)$),
 (tagy, 1, δ , TAG, $\lambda t (t=y)$),
 (tagz, 1, δ , TAG, $\lambda t (t=z)$),
 (modei, 1, δ , MODE, $\lambda m (m=i)$),
 (modeb, 1, δ , MODE, $\lambda m (m=b)$) }
 }

(continued)

Figure 1.1. An affix grammar.

This AG defines a language in which variables may be declared, at most once, to be either int or bool, and assignments are allowed between variables of the same type.

P:-

```
(p1)    block      :  var declns(L) begin stmts(L) end  .
(p2)    declns(L)  :  empty(L)  ;
(p3)                                declns(L1) tag(T) : type(M) ;
                                      declare(L1,T,M,L)  .
(p4)    stmts(L)   :  stmt(L)   ;
(p5)                                stmts(L) ; stmt(L)  .
(p6)    stmt(L)    :  vble(L,M) := vble(L,M1) equal(M1,M)  .
(p7)    vble(L,M)  :  tag(T) identify(L,T,M)  .
(p8)    tag(T)     :  x tagx(T)  ;
(p9)                                y tagy(T)  ;
(p10)                               z tagz(T)  .
(p11)    type(M)   :  int modei(M)  ;
(p12)                               bool modeb(M)  .
```

Figure 1.1 (concluded)

block

```

=> var declns(xiyi) begin stmts(xiyi) end
=> var declns(xiyi) begin stmt(xiyi) end
=> var declns(xiyi) begin vble(xiyi,i) := vble(xiyi,i)
    equal(i,i) end
=> var declns(xiyi) begin vble(xiyi,i) := vble(xiyi,i)
    end
=> var declns(xiyi) begin vble(xiyi,i) := tag(x)
    identify(xiyi,x,i) end
=> var declns(xiyi) begin vble(xiyi,i) := tag(x) end
=> var declns(xiyi) begin vble(xiyi,i) := x tagx(x) end
=> var declns(xiyi) begin vble(xiyi,i) := x end
=> var declns(xiyi) begin tag(y) identify(xiyi,y,i) :=
    x end
=> var declns(xiyi) begin tag(y) := x end
=> var declns(xiyi) begin y tagy(y) := x end
=> var declns(xiyi) begin y := x end
=> var declns(xi) tag(y) : type(i) ; declare(xi,y,i,xiyi)
    begin y := x end
=> var declns(xi) tag(y) : type(i) ; begin y := x end
=> var declns(xi) tag(y) : int model(i) ; begin y := x
    end
=> var declns(xi) tag(y) : int ; begin y := x end
=> var declns(xi) y tagy(y) : int ; begin y := x end
=> var declns(xi) y : int ; begin y := x end
=> var declns() tag(x) : type(i) ; declare(,x,i,xi) y :
    int ; begin y := x end
=> var declns() tag(x) : type(i) ; y : int ; begin y :=
    x end
=> var declns() tag(x) : int model(i) ; y : int ; begin
    y := x end
=> var declns() tag(x) : int ; y : int ; begin y := x end

```

(continued)

Figure 1.2. Derivation of a sentence in the AG of figure 1.1.

=> var declns() x tagx(x) : int ; y : int ; begin y := x end

=> var declns() x : int ; y : int ; begin y := x end

=> var empty() x : int ; y : int ; begin y := x end

=> var x : int ; y : int ; begin y := x end

Figure 1.2 (concluded)

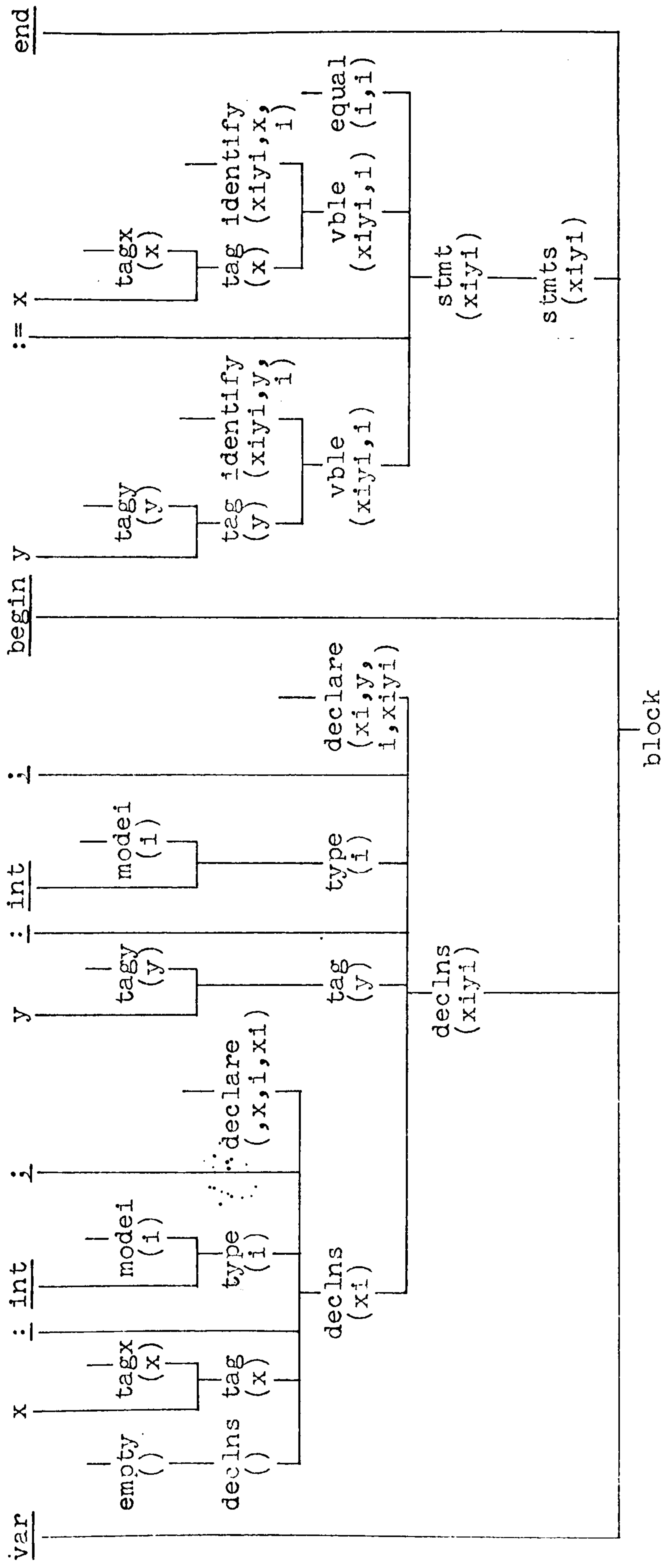


Figure 1.3 Derivation tree of a sentence in the AG of figure 1.1.

(Compare figure 1.2.)

The essential difference between our definition of an AG and that given by Koster is our introduction of affix-variables. In Koster's definition the hypernotations (or "affix expressions" as he calls them) contain affix-nonterminals rather than affix-variables; production rules are generated from meta-rules by replacing each affix-nonterminal by an affix which can be derived from it.

The distinction we have made is genuine - the variable is the object which is to be replaced, throughout a meta-rule, by a single affix; the affix-nonterminal defines the domain of that variable. VWGs, like AGs, do not make that distinction. A consequence of this in the ALGOL 68 syntax is the need to use metanotions (affix-nonterminals) like 'LMODE' and 'RMODE', which have the same domain as 'MODE' (van Wijngaarden 68). Really the reason for not making this distinction was notational and expository convenience rather than any practical or theoretical problems, as Koster himself implies in the discussion following his paper (Koster 70).

That our modification to the definition does not affect the power of AGs can be shown by a simple construction. Given an AG, unite B with A_n , add to R a new affix-rule $b : a$ for each variable b with associated affix-nonterminal a, and discard B and D. The resulting grammar conforms to Koster's definition, and its language is the same as that of the original AG.

The motivation for our modification is precisely to avoid such artificiality. Why this is important will not be clear at this stage, but will become so when we introduce extended affix grammars (chapter 3).

In examples of AGs we will use certain notational conventions. Nonterminals, primitive predicate symbols and affix-terminals will usually be represented by sequences of lower-case letters and hyphens. Affix-nonterminals will be represented by sequences of upper-case letters, and each affix-variable will usually be represented by the same sequence as its associated affix-nonterminal,

possibly followed by a digit. (Thus, in the example of section 1.0, 'MODE' and 'LMODE' were not affix-nonterminals but variables.) Terminals will usually be represented by single letters, digits or special symbols. Blanks will be used where necessary to separate adjacent symbols. If two or more affix-rules or meta-rules have identical left sides, then they may be grouped together with the common left side written once only and the right sides separated by semicolons.

Notice that derivations of sentences in an AG are similar to derivations in a CFG, with notions playing a part similar to that of nonterminals in a CFG. The essential difference is that an AG may have an infinite number of notions and an infinite number of production rules. Nevertheless, a derivation in an AG may be represented by a derivation tree as in a CFG. The root of the tree will be labelled by e , the leaves by terminal symbols, and other nodes by notions.

An example of an AG is given in figure 1.1. A derivation of a sentence in that AG is shown in figure 1.2, and the corresponding derivation tree is shown in figure 1.3.

1.2. Well-formed Affix Grammars

Our definition in section 1.1 of an AG is quite satisfactory from the point of view of language definition. Our interest, however, will be primarily in the analysis of sentences in a language. This necessitates restrictions on the class of AGs which are of interest.

Some auxiliary definitions will be useful. An occurrence of an affix-variable in an inherited affix-position on the left side, or in a derived affix-position on the right side, of a meta-rule is termed a defining occurrence of that variable. An occurrence of a variable which is not a defining occurrence is termed an applied

occurrence of that variable.

A well-formed affix grammar is one which satisfies conditions c1-c3 following.

(c1) If the hypernotation $x(f_1, \dots, f_{N_x})$ occurs in the right side of a meta-rule, then for each $i \in [1, N_x]$,

$$\tau_{x,i} = \delta \text{ implies } f_i \in B \wedge L(D(f_i)) \supseteq L(\alpha_{x,i}) ,$$

$$\tau_{x,i} = \epsilon \text{ implies } f_i \in B \wedge L(D(f_i)) \subseteq L(\alpha_{x,i}) \\ \vee f_i \in L(\alpha_{x,i}) .$$

(c2) For each $q \in Q$, suppose the inherited affix-positions of q are those numbered i_1, \dots, i_m , and the derived affix-positions of q are those numbered d_1, \dots, d_n . (Thus $m+n = N_q$.) Then there is given a total recursive function

$$\tilde{F}_q : L(\alpha_{q,i_1}) \times \dots \times L(\alpha_{q,i_m}) \rightarrow L(\alpha_{q,d_1}) \times \dots \times L(\alpha_{q,d_n}) \cup \{\omega\}$$

such that $\tilde{F}_q(y_{i_1}, \dots, y_{i_m}) = (y_{d_1}, \dots, y_{d_n})$ if and only if $F_q(y_1, \dots, y_{N_q})$ evaluates to true.

We use the term "associated function" to refer variously to F_q or to \tilde{F}_q . In each case the context should make clear which is meant.

(c3) Each affix-variable occurring in a meta-rule has exactly one defining occurrence in that meta-rule. Furthermore, if this defining occurrence is in a hypernotation Z on the right side of the meta-rule, then there is no applied occurrence of that variable in Z or in any hypernotation to the left of Z .

Condition c1 cuts off "invisible" blind alleys. Condition c2 demonstrates the primary purpose of the primitive predicates: they are functions mapping their inherited affixes on to their derived affixes.

We can best illustrate the significance of these conditions with the aid of our example of section 1.0. Consider the meta-rule

`assignment(LMODE) : destination(LMODE) check-ref(LMODE,MODE)`
`becomes-symbol source(MODE)` ,

and assume that the single affix-positions of 'assignment' and 'destination' are derived, that the affix-positions of 'check-ref' are respectively inherited and derived, and that the single affix-position of 'source' is inherited. Then the meta-rule is well-formed according to condition c3. Now consider what happens during a left-to-right goal-oriented parse of some sentence when the goal is 'assignment'. Suppose a part of the sentence has been successfully reduced to the notion 'destination(reference to real)'; then the variable 'LMODE' receives the value 'reference to real'. Now $\tilde{F}_{\text{check-ref}}(\text{reference to real})$ is evaluated, yielding the value 'real'; this implies that $\text{check-ref}(\text{reference to real}, \text{real}) \rightarrow \lambda$. Thus 'MODE' receives the value 'real'. Suppose now 'becomes-symbol' is scanned. Then, since the affix of 'source' is inherited, this affix becomes part of the new sub-goal, which is therefore 'source(real)'. When this sub-goal has been attained, the goal has also been attained. The affix of 'assignment', being derived is now made to be the value of 'LMODE', namely 'reference to real'. This completes the re-construction of the production rule applied at this stage of the parse.

Our example illustrates the fact that a defining occurrence of a variable is a position in which it receives a value, and an applied occurrence of a variable is a position in which its value is used. Condition c3 ensures that each variable has a value when required, and that no variable ever receives more than one value (which would be contrary to the generation rules for production rules).

Our example also brings out the distinction between inherited and derived affixes. In the case of a primitive predicate, they respectively the input and output parameters of its associated function. In the case of a nonterminal notion, its inherited

: those which are known before commencing a left-to-right

scan of a terminal production of that notion (and are passed down the parse tree); and its derived affixes are those which are known only after that scan (and are passed up the parse tree).

The purpose of condition c1 is to ensure that during a parse no variable ever receives an affix which is outside its domain. This might result in a parse using "production rules" which cannot legally be generated from the meta-rules of the AG, or the errant affix might end up as an input parameter of a function and cause a malfunction by being outside the domain of that parameter.

Koster (Koster 70) gives five conditions for well-formedness of an AG. The conditions we have given are equivalent to his second, third and fourth conditions. His fifth condition excludes left recursion, and was motivated solely by his own choice of parsing algorithm (top-down). His first condition prohibits a variable from occurring more than once on the left side of a meta-rule, and was designed to permit a simple transcription of the meta-rules to form the parser body. We do not include such conditions because we wish, in defining a class of AGs which are potentially suitable for parsing, to avoid reference to any particular parsing technique.

Observe that it is not possible, in general, to determine whether condition c1 holds for a given AG. This follows from the fact that it is undecidable whether the language generated by one CFG is a subset of the language generated by another CFG (Hopcroft 69, corollary 14.1). This leads us to state the following theorem.

Theorem 1.1. It is undecidable whether an arbitrary affix grammar is well-formed.

As Koster points out, this negative result is not likely to be of much significance in practice. It is to be expected that any practical AG will be written in such a way that well-formedness may

be determined by inspection.

The AG of figure 1.1 is well-formed.

1.3. Formal Properties of Affix Grammars

In this section we demonstrate some theoretical results intended to give some insight into the power of AGs as a means of defining languages.

Theorem 1.2. For every Turing machine T there exists an affix grammar which generates the language recognised by T .

Proof. We use the Turing machine formalism adopted in (Hopcroft 69), except that we represent a machine configuration by a triple (q, α, β) , in which q is the state of the machine, α is the string of symbols on the tape to the left of the tape head, and β is the string of symbols on the non-blank portion of the tape under and to the right of the tape head.

We construct an AG G which simulates the action of T . Each machine configuration (q, α, β) will be represented by the proto-notation $q(\alpha, \beta)$. A change in configuration will be represented by a derivation which simply replaces one (proto)notation by another.

The affix-terminals of G are the tape symbols of T . The affix-nonterminals are 'SYMBOL', whose terminal productions are the tape symbols, and 'LEFT' and 'RIGHT', whose terminal productions are the strings of tape symbols.

The terminals of G are the input symbols of T . The control of G contains one element

$$(q, 2, (1, 1), (\text{LEFT}, \text{RIGHT}), -)$$

each possible state q of T , plus the following:-


```

(init, 1, 1, RIGHT, - )      ,
(attachleft, 3, (1,1,δ), (LEFT,SYMBOL,LEFT),
                             λx λs λy (xs=y) )      ,
(attachright, 3, (1,1,δ), (RIGHT,SYMBOL,RIGHT),
                             λx λs λy (sx=y) )      ,
(detachleft, 3, (1,δ,δ), (LEFT,SYMBOL,LEFT),
                             λx λs λy (x=ys) )      ,
(detachright, 3, (1,δ,δ), (RIGHT,SYMBOL,RIGHT),
                             λx λs λy (x=sy) )      ,
(equal, 2, (1,1), (SYMBOL,SYMBOL), λs λt (s=t) )    ,
(blank, 1, 1, RIGHT, λx (x=λ) ) .

```

We use as affix-variables 'SYMBOL', 'SYMBOL1' (associated with the affix-nonterminal 'SYMBOL'), 'LEFT', 'LEFT1' (associated with 'LEFT'), 'RIGHT', 'RIGHT1', 'RIGHT2', 'RIGHT3' (associated with 'RIGHT').

The rules of T are transcribed into meta-rules as follows.

T-rule	$\delta(q_i, s_i) = (q_f, s_f, R)$
Meta-rule	$q_i(\text{LEFT}, \text{RIGHT}) : \text{detachright}(\text{RIGHT}, \text{SYMBOL}, \text{RIGHT1})$ $\text{equal}(\text{SYMBOL}, s_i) \text{ attachleft}(\text{LEFT}, s_f, \text{LEFT1})$ $q_f(\text{LEFT1}, \text{RIGHT1})$
Effect	$\boxed{\dots\dots s_i \mid s \mid \dots\dots} \vdash \boxed{\dots\dots s_f \mid s \mid \dots\dots}$ <div style="display: flex; justify-content: space-around; width: 100%;"> ↑ ↑ </div>
T-rule	$\delta(q_i, B) = (q_f, s_f, R)$
Meta-rule	$q_i(\text{LEFT}, \text{RIGHT}) : \text{blank}(\text{RIGHT}) \text{ attachleft}(\text{LEFT}, s_f, \text{LEFT1})$ $q_f(\text{LEFT1}, \text{RIGHT})$
Effect	$\boxed{\dots\dots \mid \text{blanks} \dots} \vdash \boxed{\dots\dots s_f \mid \text{blanks} \dots}$ <div style="display: flex; justify-content: space-around; width: 100%;"> ↑ ↑ </div>
T-rule	$\delta(q_i, s_i) = (q_f, s_f, L)$
Meta-rule	$q_i(\text{LEFT}, \text{RIGHT}) : \text{detachright}(\text{RIGHT}, \text{SYMBOL}, \text{RIGHT1})$ $\text{equal}(\text{SYMBOL}, s_i) \text{ attachright}(\text{RIGHT1}, s_f, \text{RIGHT2})$ $\text{detachleft}(\text{LEFT}, \text{SYMBOL1}, \text{LEFT1})$ $\text{attachright}(\text{RIGHT2}, \text{SYMBOL1}, \text{RIGHT3})$ $q_f(\text{LEFT1}, \text{RIGHT3})$
Effect	$\boxed{\dots\dots s \mid s_i \mid \dots\dots} \vdash \boxed{\dots\dots s \mid s_f \mid \dots\dots}$ <div style="display: flex; justify-content: space-around; width: 100%;"> ↑ ↑ </div>

T-rule	$\delta(q_i, B) = (q_f, s_f, L)$
Meta-rule	$q_i(\text{LEFT}, \text{RIGHT}) : \text{blank}(\text{RIGHT}) \text{ attachright}(\text{RIGHT}, s_f,$ $\text{RIGHT1}) \text{ detachleft}(\text{LEFT}, \text{SYMBOL}, \text{LEFT1})$ $\text{attachright}(\text{RIGHT1}, \text{SYMBOL}, \text{RIGHT2})$ $q_f(\text{LEFT1}, \text{RIGHT2})$
Effect	$\boxed{\dots s \text{blanks} \dots} \vdash \boxed{\dots s s_f \text{blanks} \dots}$ <div style="display: flex; justify-content: space-around; width: 100%;"> <div style="text-align: center;">↑</div> <div style="text-align: center;">↑</div> </div>

In addition there is the following meta-rule:

$e : \text{init}(\lambda) ,$

plus one meta-rule for each input symbol s :

$\text{init}(\text{RIGHT}) : \text{attachright}(\text{RIGHT}, s, \text{RIGHT1}) \text{ init}(\text{RIGHT1}) s ,$

plus one meta-rule for each final state q of T :

$q(\text{LEFT}, \text{RIGHT}) : \lambda ,$

plus the following meta-rule, where q_0 is the initial state of T :

$\text{init}(\text{RIGHT}) : q_0(\lambda, \text{RIGHT}) .$

We prove that a change of machine configuration corresponds to a replacement of one production by another, as stated. Consider the first of the four cases of transcription. Let $\alpha, \beta, \gamma, \eta$ be arbitrary strings of tape symbols, and let s be an arbitrary tape symbol. Substituting α for LEFT , β for LEFT1 , γ for RIGHT , η for RIGHT1 and s for SYMBOL in the meta-rule

$q_i(\text{LEFT}, \text{RIGHT}) : \text{detachright}(\text{RIGHT}, \text{SYMBOL}, \text{RIGHT1})$
 $\dots \text{equal}(\text{SYMBOL}, s_i) \text{ attachleft}(\text{LEFT}, s_f, \text{LEFT1})$
 $q_f(\text{LEFT1}, \text{RIGHT1}) ,$

we obtain the production rule

$q_i(\alpha, \gamma) \rightarrow \text{detachright}(\gamma, s, \eta) \text{ equal}(s, s_i) \text{ attachleft}(\alpha, s_f,$
 $\beta) q_f(\beta, \eta) .$

Now, $\text{detachright}(\gamma, s, \eta) \rightarrow \lambda$ if and only if $\gamma = s\eta$;

$\text{equal}(s, s_i) \rightarrow \lambda$ if and only if $s = s_i$;

and $\text{attachleft}(\alpha, s_f, \beta) \rightarrow \lambda$ if and only if $\alpha s_f = \beta$.

Thus, if $\delta(q_i, s_i) = (q_f, s_f, R)$, then $q_f(\alpha s_f, \eta)$ is a production

of $q_i(\alpha, s_i, \eta)$; moreover the sequence of direct productions comprising this production includes only one replacement of a nonterminal notion. Exhaustive testing shows that "only if" applies here as well as "if". But, by definition of T , $(q_i, \alpha, s_i, \eta) \vdash (q_f, \alpha s_f, \eta)$ if and only if $\delta(q_i, s_i) = (q_f, s_f, R)$. Similar arguments apply in the other three cases of transcription. Thus, in general, $q_2(\alpha_2, \beta_2)$ is a production of $q_1(\alpha_1, \beta_1)$ involving exactly one replacement of a nonterminal notion if and only if $(q_1, \alpha_1, \beta_1) \vdash (q_2, \alpha_2, \beta_2)$. It follows by induction that $q_1(\alpha_1, \beta_1) \Rightarrow^* q_2(\alpha_2, \beta_2)$ if and only if $(q_1, \alpha_1, \beta_1) \vdash^* (q_2, \alpha_2, \beta_2)$. In particular, $q_0(\lambda, \tau) \Rightarrow^* q(\alpha, \beta)$ if and only if $(q_0, \lambda, \tau) \vdash^* (q, \alpha, \beta)$.

It can further be shown that, for any string τ of input symbols, $e \Rightarrow \text{init}(\lambda) \Rightarrow^* \text{init}(\tau)\tau \Rightarrow q_0(\lambda, \tau)\tau$. Also $q(\alpha, \beta) \rightarrow \lambda$ if and only if q is a final state of T . Thus $e \Rightarrow^* q_0(\lambda, \tau)\tau \Rightarrow^* q(\alpha, \beta)\tau \Rightarrow \tau$ if and only if q is a final state of T and $(q_0, \lambda, \tau) \vdash^* (q, \alpha, \beta)$. That is, τ is a sentence of G if and only if T accepts τ .

Corollary 1.3. Every recursively enumerable set can be generated by an affix grammar.

Note that the AG which simulates the Turing machine is well-formed according to our definition (but not according to Koster's definition). The foregoing results are more of theoretical than of practical interest, however, as we are primarily interested in those AGs from which parsers can be constructed. Since the recognition problem is solvable only for recursive sets (Hopcroft 69), it follows that every AG of interest to us generates a recursive set.

1.4. Some Comments on Affix Grammars

The primary advantage of AGs is their suitability for syntax-directed parsing. This is a consequence of their being an extension of CFGs, which have been extensively studied and for which parsing techniques have been developed to a high degree of refinement. Koster's extension was designed to allow many of these techniques, suitably generalised, to be applied to AGs as well. This possibility has been demonstrated by Koster himself, who describes a top-down parsing method (Koster 70), and also by Crowe, who gives a bottom-up method (Crowe 72). In chapter 2 we shall provide a further demonstration by describing a third technique.

The power of AGs depends very largely on the functions associated with its primitive predicate symbols. The definition of an AG in fact permits functions of arbitrary complexity to be included. Indeed there is nothing to exclude an AG which has a meta-rule

e : string(STR) acceptable(STR) ,

and in which the whole recognition problem is solved by one "primitive" predicate function! More seriously, it is noteworthy that in the AG which simulates a Turing machine (theorem 1.2), all the functions were simple to the point of triviality. So also were most of the functions in our example AG of figure 1.1. This strongly suggests that AGs may contain power in excess of that which is strictly necessary to define languages. In practice this can be an advantage, facilitating the determination of complex relationships among affixes.

There is no defined mechanism for specifying the primitive predicate functions. The simple functions occurring in our examples we have defined in λ -notation. In a practical AG, intended to specify the action of a parser, the functions might well be expressed in a suitable programming language. This flexibility is valuable in practice, although it does require that a constructor for AGs must be used in conjunction with a compiler.

It is noteworthy that, although affixes are formally defined as strings of symbols, their nature has no real relevance to the essential structure of AGs. In fact the only role of the affix-terminals, -nonterminals and -rules is to define the domains of the affix-variables and of the affix-positions. Even this information is of limited usefulness, from a formal point of view, since by theorem 1.1 it cannot be used automatically to check that an AG is well-formed. There is no reason why affixes should not be allowed to be objects of any desired type. Koster himself gives an example in which some affixes are integers or arrays (Koster 70). This flexibility again is valuable in practice, but it does suggest a degree of over-definition of AGs. The affix-terminals, -nonterminals and -rules could well be dispensed with, and the domains of affix-variables and of affix-positions could then be arbitrary sets. The absence of a defined mechanism for specifying these domains would have the same advantages and disadvantages as the lack of a defined mechanism for specifying the primitive predicate functions, since these functions are closely tied to the nature of the affixes anyway.

As we have already seen, an AG tends to contain many primitive predicates, often rather trivial, and its meta-rules tend to become rather cluttered with them. As a result, AGs tend to be tedious to write, and subsequently difficult to read. From the point of view of providing a language definition method suitable for human readers, this is perhaps the most serious disadvantage of AGs.

In an attempt to remedy some of the drawbacks discussed here, to make the grammars easier to write and to read, without sacrificing the applicability of syntax-directed parsing techniques, we propose in chapter 3 a modified form of two-level grammar, based however on AGs. But first we turn our attention in chapter 2 to the parsing problem for AGs.

CHAPTER 2LEFT-TO-RIGHT PARSERS FOR WELL-FORMED AFFIX GRAMMARS

2.1. LR(k) Parsing for Context-Free Grammars

Knuth (Knuth 65) defined an LR(k) grammar (where $k \geq 0$) to be a CFG satisfying the condition that, for each (right) canonical form $\phi v \tau$, the first production rule $N \rightarrow v$ of the canonical parse of $\phi v \tau$ can be determined uniquely from ϕv and the first k symbols of τ . Thus for an LR(k) grammar there is a parser which, starting with a sentence, repeatedly determines from the current canonical form $\phi v \tau$ the relevant production rule $N \rightarrow v$ and replaces v by N (this is called a reduction) to obtain a new canonical form $\phi N \tau$; the parse is complete when the new form is S , the distinguished nonterminal of the grammar. Since a right canonical derivation is a right-to-left process, and a parse is the reverse of a derivation, this parser will scan the sentence from left to right, except for look-aheads of up to k symbols. In fact, as Knuth pointed out, the LR(k) grammars are the largest class of CFGs whose sentences can be parsed deterministically from left to right. Also, every LR(k) grammar is unambiguous.

DeRemer (DeRemer 69) showed how to construct efficient parsers from LR(k) grammars. As his work forms the basis of much of the material in this chapter, we summarise his main results here. Consider the CFG $G = (V_T, V_N, S, P)$. We assume that the distinguished nonterminal S does not occur on the right side of any production rule in P . We also assume an arbitrary numbering of the production rules.

Suppose $\phi v \tau$ is a canonical form which has a canonical derivation $S \Rightarrow^* \phi N \tau \Rightarrow \phi v \tau$, and suppose $N \rightarrow v$ is the p -th production rule in P . Let $\#_p$ be a special symbol, not in $V_N \cup V_T$, uniquely associated with this production rule. Then $\phi v \#_p$ is a characteristic string of $\phi v \tau$. Knuth's definition can thus be re-stated: G is LR(k) if and only if every canonical form $\beta \tau$ of G , except S , has a unique characteristic string $\beta \#_p$ which can be determined by investigating only β and the first k symbols of τ .

DeRemer proved that the set of characteristic strings of G is the regular language generated by the grammar $G_C = (V_T', V_N', S', P')$, where

$$\begin{aligned} V_T' &= V_T \cup V_N \cup \{\#_0, \#_1, \dots\} \quad , \\ V_N' &= \{ N' \mid N \in V_N \} \quad , \\ P' &= \{ N' \rightarrow v \#_p \mid N \rightarrow v \text{ is the } p\text{-th production rule in } P \} \\ &\quad \cup \{ N' \rightarrow v M' \mid N \rightarrow v M \mu \text{ is in } P, \text{ and } M \in V_N \} \quad . \end{aligned}$$

G_C is called the characteristic grammar of G . The characteristic finite-state machine (CFSM) of G is the reduced deterministic finite-state machine which accepts the sentences of G_C , i.e. the characteristic strings of G . The CFSM can be constructed from G_C (Hopcroft 69); DeRemer also gives a method of constructing the CFSM directly from G (DeRemer 71).

The CFSM contains transitions under terminals, nonterminals and $\#$ -symbols. A read state is a state of the CFSM out of which there are only terminal- and nonterminal-transitions; a reduce state is one out of which there is exactly one $\#$ -transition and no terminal transitions; an inadequate state is one which is neither a read state nor a reduce state. DeRemer proved that a CFG is LR(0) if and only if its CFSM contains no inadequate states. (Actually, his definition in (DeRemer 69) of reduce states precluded any nonterminal transitions out of them, and his proof of the above result assumed the latter definition. In (DeRemer 71) he modified the definition to allow one nonterminal transition out of a reduce state. In fact, any number of such transitions may be allowed without affecting the result.)

DeRemer then showed that the sentences of an LR(0) grammar G can be parsed by the following algorithm based on G 's CFSM. This algorithm, which is called the LR(0) parsing algorithm, uses a stack on which are stored the names of states of the CFSM.

- Step 1. Start the CFSM in its initial state, and initialise the stack to be empty. Be prepared to start reading the input string from its first symbol.
- Step 2. Stack the name of the current state. If the current state is a read state, go to step 3. If the current state is a reduce state, consider the $\#$ -transition out of this state, and go to step 4.
- Step 3. (Read state). Read the next terminal from the input string. If there is a transition out of the current state under that terminal, then change to the state at the end of that transition, and go to step 2. Otherwise, the input string is not a sentence of G .
- Step 4. (Reduce state). Let $N \rightarrow v$ be the production rule associated with the $\#$ -symbol on the considered transition, and let n be the number of symbols in v . Pop n state names from the stack. If $N=S$, the parse has been successfully completed. Otherwise, there will be a transition under N out of the state whose name is now at the top of the stack. Change to the state at the end of that transition and go to step 2.

Figure 2.1 gives an example of an LR(0) grammar together with its CFSM and a history of the LR(0) parser when applied to a sentence of the grammar.

In a CFSM all transitions into a given state are under the same (terminal or nonterminal) symbol (DeRemer 71). Thus each state on the stack, with the exception of the initial state,

uniquely identifies the symbol which caused that state to be accessed. Thus the contents of the stack are a representation of the prefix of the current canonical form which has already been scanned.

Inadequate states, which are likely to appear in the CFSMs of most grammars of interest, cannot be handled by the LR(0) parsing algorithm because in such a state there is a choice between reading a new terminal and making a reduction, and/or a choice among several reductions. In many cases this local non-determinism can be resolved by a look-ahead in the input string. In the following we assume that P contains production rules of the form $S \rightarrow \sigma \dashv^k$, where $\dashv \in V_T$, and neither S nor \dashv occurs in any other production rule.

With each terminal- and #-transition out of an inadequate state q is associated a k-symbol look-ahead set: for a transition under a terminal t this is

$$LAS_k(q, t) = \{ t\beta \in V_T^k \mid S \Rightarrow^* \phi t \beta \tau, \text{ and } \phi \text{ accesses } q \text{ in the CFSM} \};$$

and for a transition under a #-symbol this is

$$LAS_k(q, \#_p) = \{ \beta \in V_T^k \mid S \Rightarrow^* \phi N \beta \tau \Rightarrow \phi \nu \beta \tau, \text{ and } \phi \nu \text{ accesses } q \text{ in the CFSM, and } N \rightarrow \nu \text{ is production rule } p \}.$$

DeRemer defined a grammar to be LALR(k) if and only if, for every inadequate state in its CFSM, the k-symbol look-ahead sets associated with the terminal- and #-transitions out of that state are mutually disjoint; each such state is called a look-ahead state.

We do not know of any algorithm for computing these look-ahead sets exactly, but DeRemer has shown (DeRemer 69) how to compute approximations to these sets. DeRemer's method computes what he calls simple look-ahead sets, each of which subsumes the corresponding (exact) look-ahead set. If in every inadequate state the simple k-symbol look-ahead sets are mutually disjoint, the grammar is called simple LR(k) (SLR(k)).

The LR(0) parsing algorithm can be made into an LALR(k) parsing algorithm by allowing it to look ahead k symbols. Step 2 is modified by the addition of the sentence "If the current state is a look-ahead state, go to step 5.". A new step is added:

Step 5. Examine the next k terminals of the input string, but do not read them. If this string is not in any of the k-symbol look-ahead sets associated with transitions out of the current state, then the input string is not a sentence of G. Otherwise, the string must be in exactly one of the look-ahead sets. If the transition associated with this set is a terminal-transition, then read a terminal from the input string (which will always match the terminal on the transition), change to the state at the end of the transition, and go to step 2. If the transition is a #-transition, then consider this transition and go to step 4.

An important point made by DeRemer is that in general some inadequate states are "more" inadequate than others. A state may be considered to be k-look-ahead if k is the smallest integer for which the k-symbol look-ahead sets associated with transitions out of that particular state are mutually disjoint. (Reduce states may be considered to be 0-look-ahead from this point of view.) The grammar is then LALR(k) if the "most" inadequate state of its CFSM is k-look-ahead. The LALR(k) parsing algorithm can easily be generalised to take advantage of this by shortening look-aheads wherever possible.

Figure 2.2 shows an LALR(1) grammar and its CFSM with the 1-symbol look-ahead sets associated with transitions out of its inadequate states.

The treatment of general LR(k) grammars is an order more complex. The CFSM of an LR(k) grammar which is not LALR(k) will contain inadequate states for which the k-symbol look-ahead sets are not mutually disjoint. Intuitively, the reason for this is

that each state of the CFSM represents an equivalence class of possible left contexts and as such may not convey to the parser enough information about the left context to enable a parsing decision to be made (even with look-ahead). DeRemer showed how states of the CFSM can be "split" (where necessary), thus partitioning the equivalence classes, to convey more detailed information about the left context, in such a way that every inadequate state of the resulting LRkCFSM is at worst k-look-ahead if the grammar is LR(k). The LR(k) parsing algorithm uses the LRkCFSM but is otherwise identical to the LALR(k) parsing algorithm.

Figure 2.3 gives an example of an LR(1) grammar, which is not LALR(k) for any k, its CFSM with associated 1-symbol look-ahead sets, and its LR1CFSM with associated 1-symbol look-ahead sets.

In practice, very many grammars of interest are LALR(1). For example, DeRemer showed that the LALR(1) grammars include the weak precedence grammars, which in turn include the simple precedence grammars. Practical grammars which are LALR(k) but not LALR(1) are likely to have CFSMs with only a small proportion of states which are "worse" than 1-look-ahead. (In the case of programming languages, this is linked to the practical necessity of programs being intelligible to humans, reading from left to right, as well as to machines.) Equally, practical grammars which are LR(k) but not LALR(k) are likely to have CFSMs in which only a few states require to be split.

These observations are of the greatest practical importance. The complexity of the computation of the k-symbol look-ahead sets increases rapidly with k. State-splitting is also computationally complex. DeRemer's approach allows these computations to be performed only when strictly necessary, and the resulting parsers are economic in terms of both size and speed.

An important advantage of LR(k) parsing in general is early detection of errors. An error is discovered immediately the offending terminal is read; or perhaps even earlier, if a look-ahead fails.

$S \rightarrow a A$	$(\#_0)$	$S \rightarrow b B$	$(\#_3)$
$A \rightarrow c A$	$(\#_1)$	$B \rightarrow c B$	$(\#_4)$
$A \rightarrow d$	$(\#_2)$	$B \rightarrow d$	$(\#_5)$

Figure 2.1 (a) The production rules of an LR(0) grammar, with associated #-symbols.

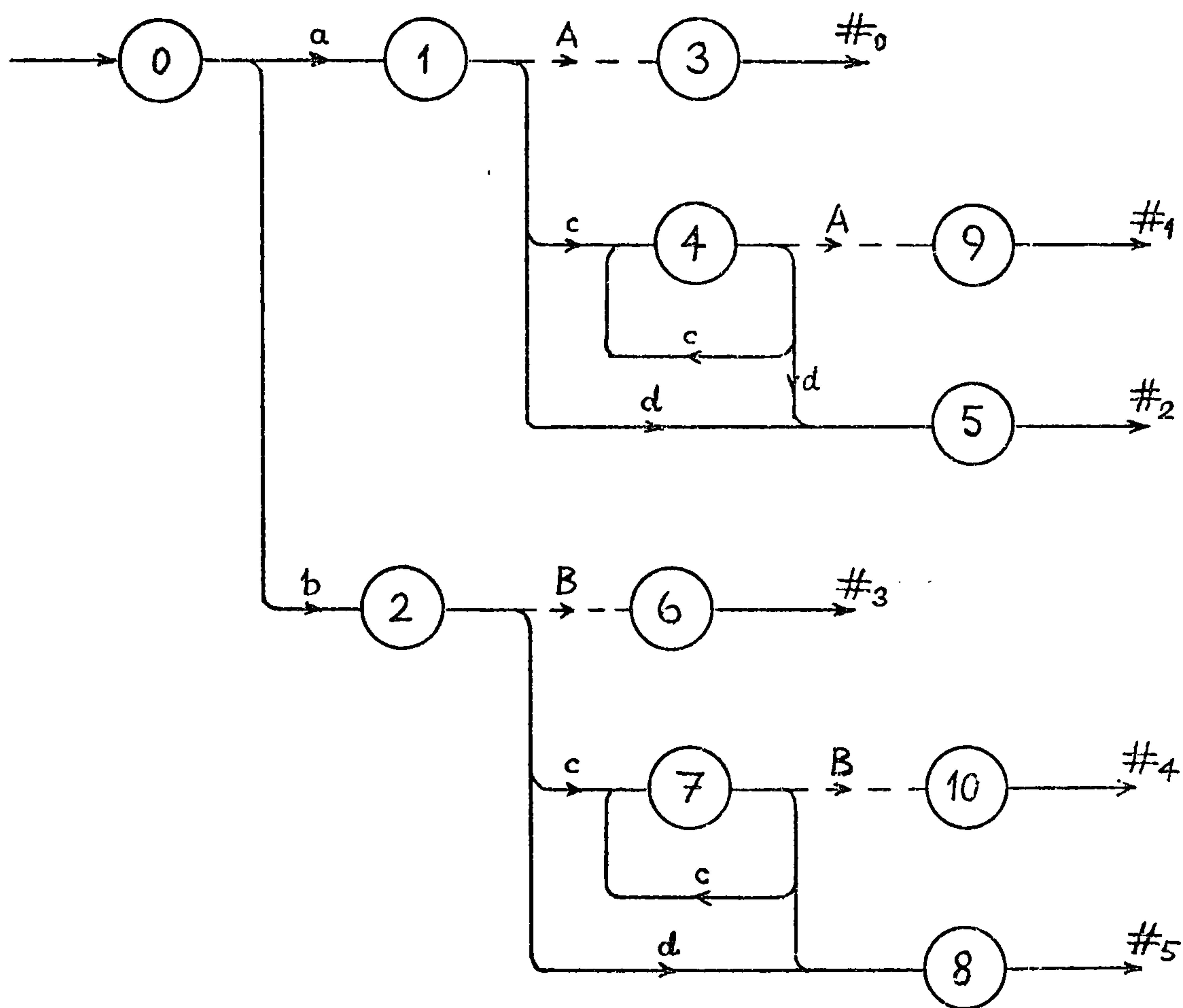


Figure 2.1(b) The CFSM of the above grammar.

(All #-transitions go to a unique state. This state is not shown, for the sake of clarity, and also because it plays no part in the parser.)

State	Stack	Remaining input string	Reduction
0		a c d	
1	0	c d	
4	0 1	d	
5	0 1 4		2 A -> d
9	0 1 4		1 A -> cA
3	0 1		0 S -> aA

Figure 2.1(c) History of the above grammar's LR(0) parser when applied to the string 'acd'.

$S \rightarrow E \cdot$	$(\#_0)$	$T \rightarrow T * P$	$(\#_3)$
$E \rightarrow E - T$	$(\#_1)$	$T \rightarrow P$	$(\#_4)$
$E \rightarrow T$	$(\#_2)$	$P \rightarrow i$	$(\#_5)$
		$P \rightarrow (E)$	$(\#_6)$

Figure 2.2 (a) Production rules of an LALR(1) grammar, with associated #-symbols.

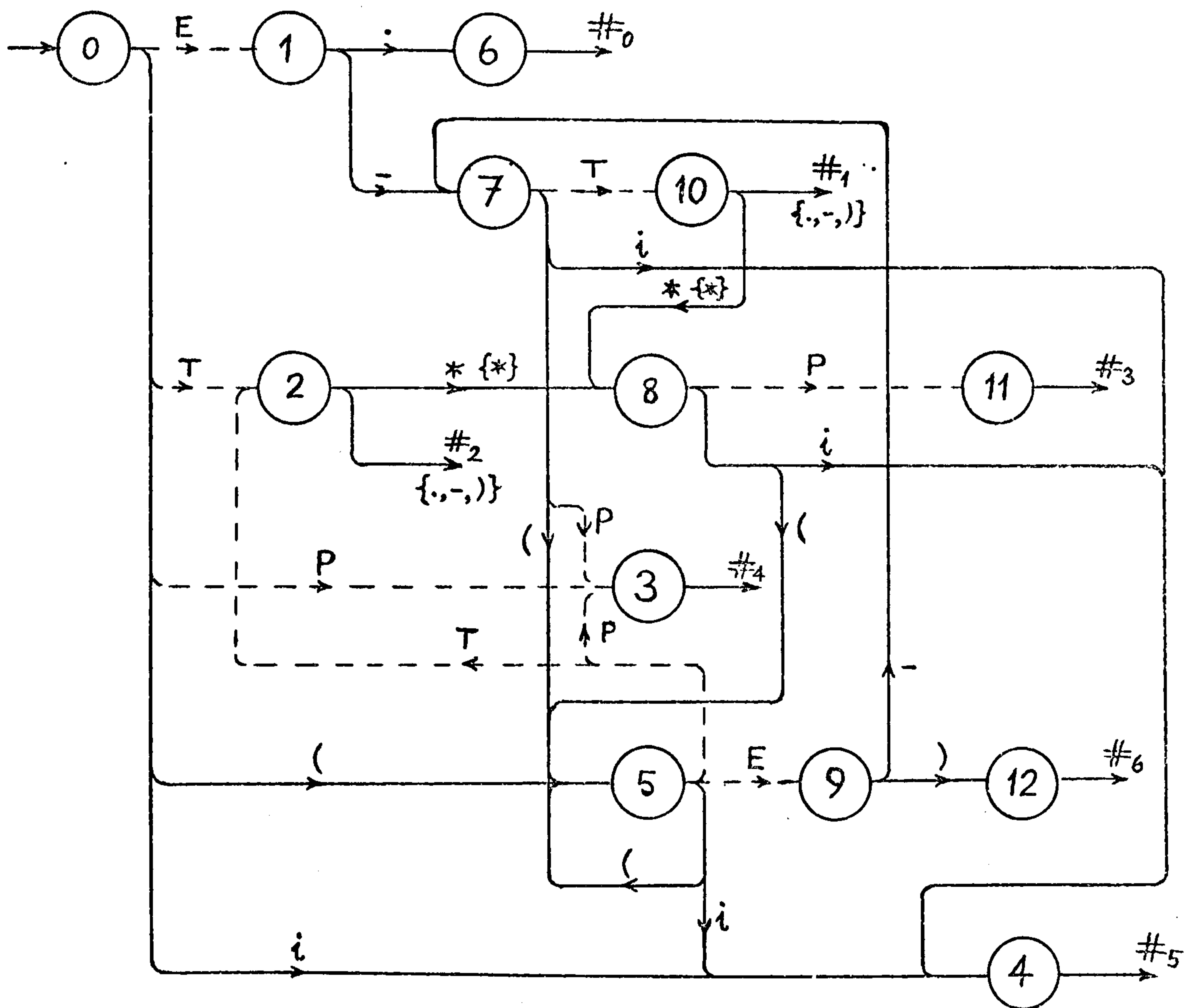


Figure 2.2 (b) CFSM of the above grammar, with 1-symbol look-ahead sets associated with transitions out of inadequate states (2 and 10).

State	Stack	Remaining input string	Look-ahead string	Reduction
0		i * i - i .		
4	0	* i - i .		5 P → i
3	0	* i - i .		4 T → P
2	0	* i - i .		
8	0 2	i - i .	*	
4	0 2 8	- i .		5 P → i
11	0 2 8	- i .		3 T → T*P
2	0	- i .		2 E → T
1	0	- i .	-	
7	0 1	i .		
4	0 1 7	.		5 P → i
3	0 1 7	.		4 T → P
10	0 1 7	.	.	1 E → E-T
1	0	.		
6	0 1			0 S → E.

Figure 2.2 (c) History of the above grammar's LALR(1) parser when applied to the string
i * i - i .

$S \rightarrow a A d . \quad (\#_0)$	$A \rightarrow e A \quad (\#_4)$
$S \rightarrow a B c . \quad (\#_1)$	$A \rightarrow e \quad (\#_5)$
$S \rightarrow b A c . \quad (\#_2)$	$B \rightarrow e B \quad (\#_6)$
$S \rightarrow b B d . \quad (\#_3)$	$B \rightarrow e \quad (\#_7)$

Figure 2.3 (a) Production rules of an LR(1) grammar, with associated #-symbols.

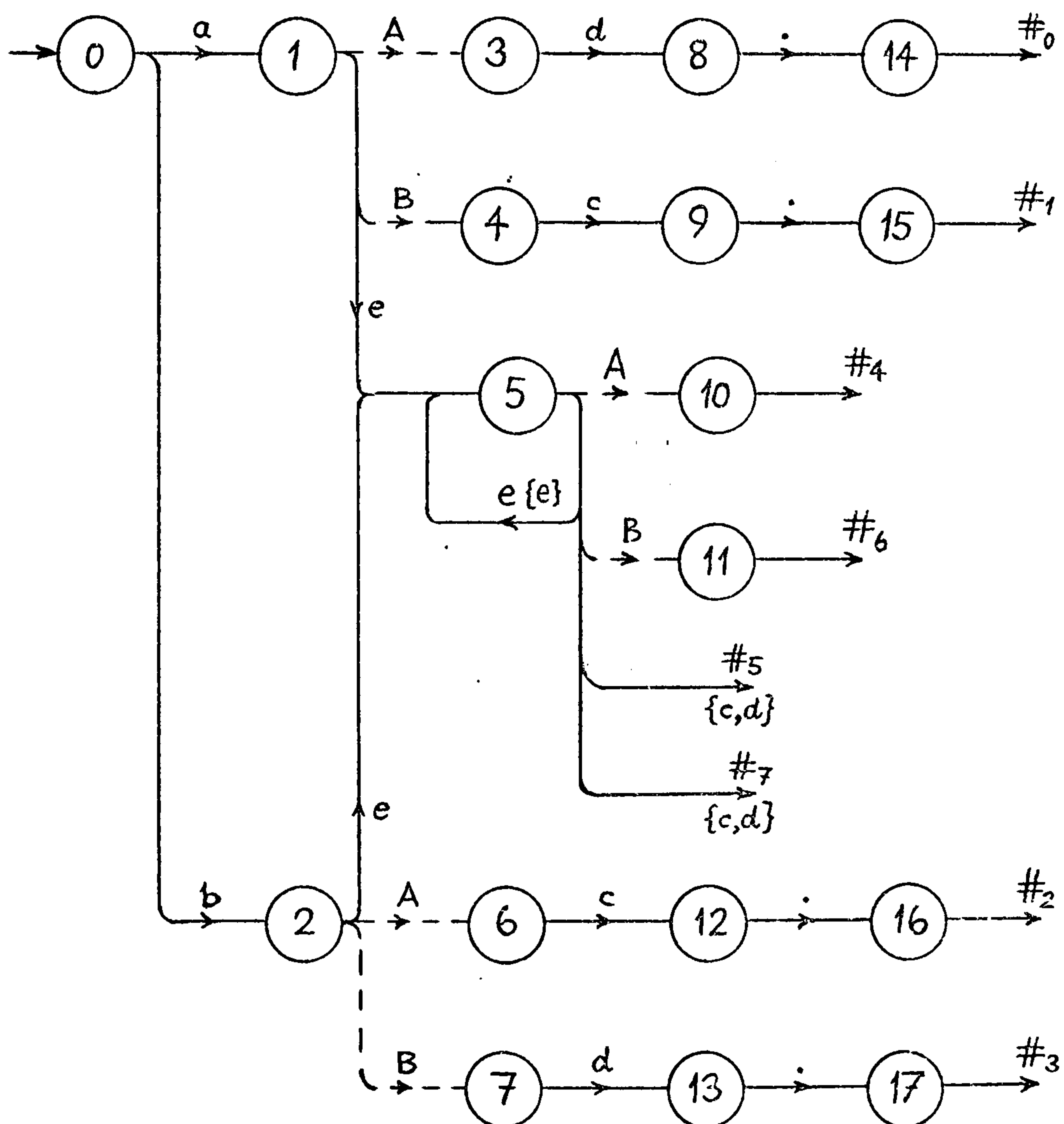


Figure 2.3 (b) CFSM of the above grammar, with 1-symbol look-ahead sets. Note that state 5 is not 1-look-ahead; in fact it is not k-look-ahead for any k.

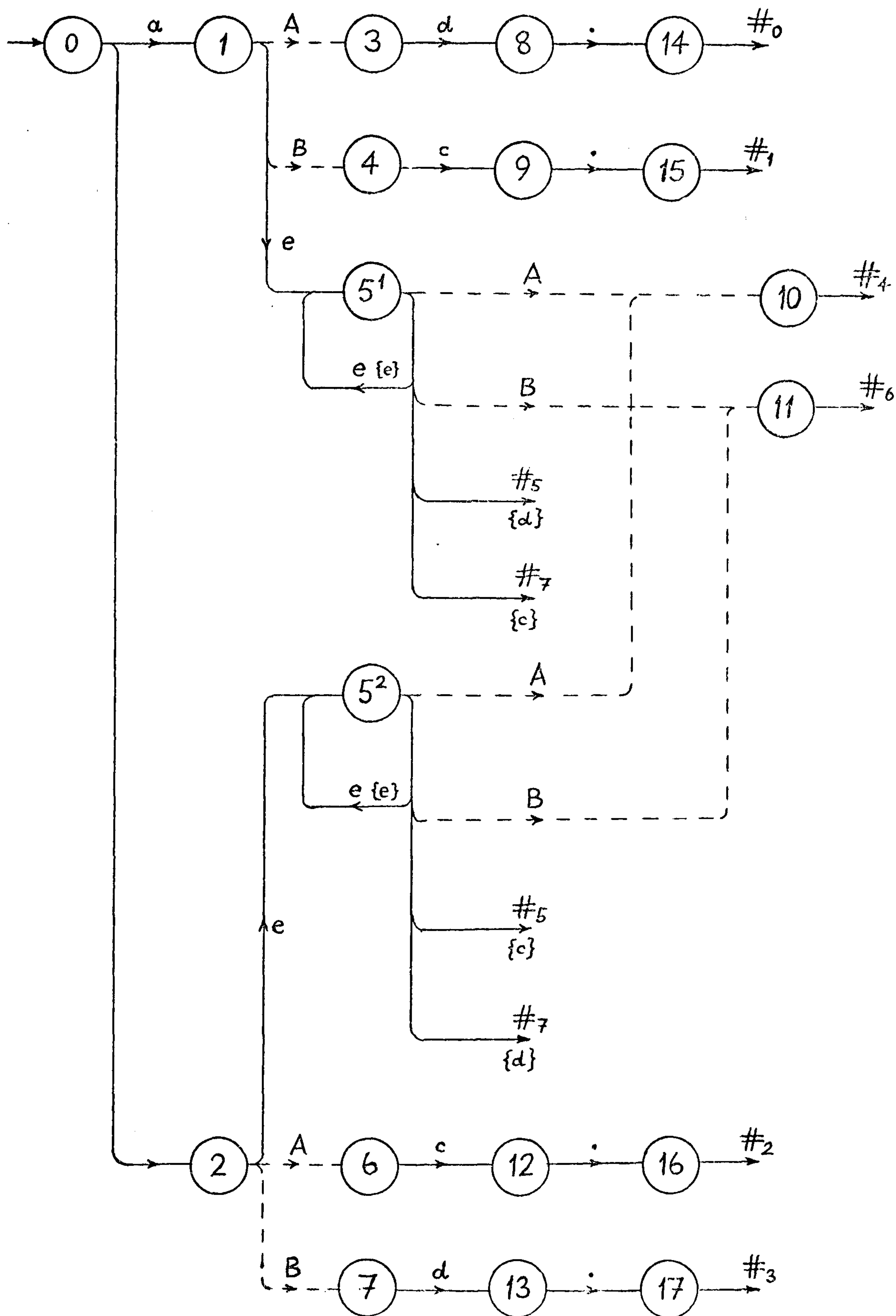


Figure 2.3 (c) CFSM of the above grammar, with 1-symbol look-ahead sets, after splitting of state 5.

In conclusion, LR(k) parsing is more general than any other deterministic CF parsing technique, has good error detection capability, and in practice is competitive in terms of size and speed. These features make the LR(k) parsing technique a good basis for extension to the parsing of languages defined by affix grammars.

2.2. Canonical Forms of an Affix Grammar

Before proceeding to the parsing problem for AGs, we introduce some notational conventions which will facilitate our arguments. Firstly, observing that the order of the affix-positions of each nonterminal or primitive predicate symbol has no bearing on the language generated by the AG nor on its "structure", we can assume without loss of generality that all the inherited affix-positions of each nonterminal or primitive predicate symbol must precede all its derived affix-positions. Secondly, we consider an affix to be an atomic object, so that we can write fg to denote a sequence of two affixes, f and g , and not the concatenation of two strings of affix-terminals. We emphasise, however, that these are merely notational conveniences and do not apply outside this chapter.

Following these conventions, we shall write a hypernotation in the form $x(\theta;K)$, where $\theta \in (BuL)^*$ is a sequence of the affixes and affix-variables occupying the inherited affix-positions of the hypernotation, and $K \in (BuL)^*$ is a similar sequence for the derived affix-positions. Note that either θ or K or both may be the empty sequence (λ). In particular, the notion e is written as $e(\lambda;\lambda)$.

We shall apply a similar convention to the parameters of the functions associated with each primitive predicate symbol q : we write $q(\alpha;\beta) \rightarrow \lambda$ or $F_q(\alpha,\beta)$ or $F_q(\alpha)=\beta$ (all of which are equivalent), where $\alpha, \beta \in L^*$.

Consider the well-formed AG

$$G = (V_n, V_t, A_n, A_t, Q, e, R, B, D, S, P) \quad .$$

Recall that L is the set of affixes of G . Let N be the set of protonotions of G ; in our new notation,

$$N \subseteq \{ x(\alpha; \beta) \mid x \in V_n \cup Q \wedge \alpha, \beta \in L^* \} \quad .$$

Consider a canonical derivation in G :-

$$\begin{aligned} e(\lambda; \lambda) & \\ \Rightarrow \pi_1 x_1(\alpha_1; \beta_1) \pi_1' & \quad (\text{applying a production rule } e(\lambda; \lambda) \rightarrow \pi_1 x_1(\alpha_1; \beta_1) \pi_1') \\ \Rightarrow^* \pi_1 x_1(\alpha_1; \beta_1) \tau_1 & \quad (\text{assuming that } \pi_1' \Rightarrow^* \tau_1) \\ \Rightarrow \pi_1 \pi_2 x_2(\alpha_2; \beta_2) \pi_2' \tau_1 & \quad (\text{applying } x_1(\alpha_1; \beta_1) \rightarrow \pi_2 x_2(\alpha_2; \beta_2) \pi_2') \\ \Rightarrow^* \pi_1 \pi_2 x_2(\alpha_2; \beta_2) \tau_2 \tau_1 & \quad (\text{assuming that } \pi_2' \Rightarrow^* \tau_2) \\ \vdots & \\ \Rightarrow^* \pi_1 \dots \pi_m x_m(\alpha_m; \beta_m) \tau_m \dots \tau_1 & \\ \Rightarrow \pi_1 \dots \pi_m \pi \tau_m \dots \tau_1 & \quad (\text{applying } x_m(\alpha_m; \beta_m) \rightarrow \pi) \end{aligned}$$

In this, $\pi, \pi_1, \dots, \pi_m, \pi_1', \dots, \pi_m' \in (N \cup V_t)^*$; $\tau_1, \dots, \tau_m \in V_t^*$; $x_1, \dots, x_{m-1} \in V_n$; $x_m \in V_n \cup Q$; and $\alpha_1, \dots, \alpha_m, \beta_1, \dots, \beta_m \in L^*$. At the last step of the derivation, if $x_m \in Q$, then $\pi = \lambda$ and $F_{x_m}(\alpha_m, \beta_m) = \underline{\text{true}}$.

This canonical derivation was quite general. This leads to the following theorem.

Theorem 2.1. Every canonical form of the affix grammar G is of the form

$$\phi = \pi_1 \dots \pi_m \pi \tau \quad ,$$

- where (i) $\pi_1, \dots, \pi_m, \pi \in (N \cup V_t)^*$ and $\tau \in V_t^*$;
(ii) for each $i \in [1, m]$, $x_{i-1}(\alpha_{i-1}; \beta_{i-1}) \rightarrow \pi_i x_i(\alpha_i; \beta_i) \pi_i'$ is a production rule for some $\pi_i' \in (N \cup V_t)^*$;
(iii) $x_0 = e$ and $\alpha_0 = \beta_0 = \lambda$;
and (iv) $x_m(\alpha_m; \beta_m) \rightarrow \pi$ is a production rule.

Proof. We prove the theorem by induction on the number of steps in the canonical derivation. The theorem is true for a

derivation of one step, since such a derivation must be of the form $e(\lambda;\lambda) \Rightarrow \sigma$, where $e(\lambda;\lambda) \rightarrow \sigma$ is a production rule.

Suppose $e(\lambda;\lambda) \Rightarrow^* \phi$ is a derivation of n steps. Consider the $(n+1)$ -th step. Suppose further that the rightmost production in ϕ occurs in π_k (where $1 \leq k \leq m$), i.e. that $\pi_{k+1}, \dots, \pi_m, \pi \in V_t^*$ but $\pi_k \notin V_t^*$. Let $\pi_k = \pi' x'(\alpha'; \beta') \gamma$, where $\pi' \in (N \cup V_t)^*$, $\gamma \in V_t^*$, and $x'(\alpha'; \beta') \in N$. Then the $(n+1)$ -th step of the canonical derivation is

$$\begin{aligned} \phi &= \pi_1 \dots \pi_{k-1} \pi' x'(\alpha'; \beta') \gamma \pi_{k+1} \dots \pi_m \pi \tau \\ &\Rightarrow \pi_1 \dots \pi_{k-1} \pi' \pi'' \gamma \pi_{k+1} \dots \pi_m \pi \tau \\ &\quad \text{(applying } x'(\alpha'; \beta') \rightarrow \pi'') \\ &= \phi' \end{aligned}$$

ϕ' has the same form as ϕ , with π_k replaced by π' , π by π'' , τ by $\gamma \pi_{k+1} \dots \pi_m \pi \tau$, and m by k .

Similar logic applies in the case $\pi_k \in V_t^*$. Thus the theorem is proved.

2.3. Auxiliary Grammar of a Well-Formed Affix Grammar

In section 2.2 we viewed an AG primarily as a generative system, that is a system for deriving sentences. No clear distinction was made between inherited and derived affixes. In this and subsequent sections, we approach the problem of parsing sentences in the AG.

Condition c3 in the definition of a well-formed AG (section 1.2) implies that, during a left-to-right scan of a sentence, the inherited affixes of a notion are known before, and its derived affixes only after, scanning the substring of the sentence which is the terminal production of that notion. Our left-to-right parser will therefore adopt the following strategy: before scanning the substring, the notion's inherited affixes are stored at the top of a special affix

stack; after scanning the substring, the notion's derived affixes are determined and placed in the affix stack immediately above its inherited affixes (replacing any affixes placed there during the parsing of the substring). Since the meta-rules of the AG do not reflect this essential distinction between the treatments of inherited and derived affixes, we must construct from the AG a new grammar which is "loosely equivalent" to the AG and which will define the action of our left-to-right parser.

Consider the well-formed AG

$$G = (V_n, V_t, A_n, A_t, Q, e, R, B, D, S, P) \quad .$$

We define an auxiliary grammar of G to be a 12-tuple

$$G_A = (V_n, V_t, A_n, A_t, Q, I, e, R, B, D, S, P_A) \quad ,$$

where I is a new set of symbols, which we call copy-symbols, and P_A is a new set of meta-rules, constructed from the meta-rules of P as described in the following paragraph. I is disjoint from $V_n \cup V_t \cup Q$.

For each meta-rule in P

$$x(\theta; \kappa) : \tau_0 x_1(\theta_1; \kappa_1) \tau_1 \dots x_n(\theta_n; \kappa_n) \tau_n \quad ,$$

where $x \in V_n$, $x_1, \dots, x_n \in V_n \cup Q$, $\tau_0, \tau_1, \dots, \tau_n \in V_t^*$, $\theta, \kappa \in B^*$, and $\theta_1, \dots, \theta_n, \kappa_1, \dots, \kappa_n \in (B \cup L)^*$, P_A will contain the meta-rule

$$\begin{pmatrix} x \\ \theta \kappa \end{pmatrix} : \begin{pmatrix} \tau_0 \iota_1 x_1 \tau_1 \dots \iota_n x_n \tau_n \\ \theta_1 \kappa_1 \dots \theta_n \kappa_n \end{pmatrix} \quad ,$$

in which ι_1, \dots, ι_n are copy-symbols, are distinct from one another, and occur only in this meta-rule. In the special case $\theta_i = \lambda$, however, we replace ι_i by λ . In addition to this meta-rule, we add to P_A a completely new meta-rule for each $i \in [1, n]$ such that $\theta_i \neq \lambda$:-

$$\begin{pmatrix} \theta \theta_1 \kappa_1 \dots \theta_{i-1} \kappa_{i-1} \theta_i \\ \theta \theta_1 \kappa_1 \dots \theta_{i-1} \kappa_{i-1} \end{pmatrix} : \begin{pmatrix} \theta \theta_1 \kappa_1 \dots \theta_{i-1} \kappa_{i-1} \\ \theta \theta_1 \kappa_1 \dots \theta_{i-1} \kappa_{i-1} \end{pmatrix} \quad .$$

For example, the AG meta-rule

$$v(A; B D) : x(\lambda; D) \text{ t } w(A; B E)$$

would be transformed into the auxiliary grammar meta-rules

$$\begin{pmatrix} v \\ A B D \end{pmatrix} : \begin{pmatrix} x \ t \ \iota \ w \\ A D A B E \end{pmatrix},$$

$$\begin{pmatrix} \iota \\ A D A \end{pmatrix} : \begin{pmatrix} \\ A D \end{pmatrix}.$$

From our transformations we can make the following observations about the meta-rules in P_A .

(1) Every meta-rule in P_A is of the form

$$\begin{pmatrix} s \\ \theta \kappa \end{pmatrix} : \begin{pmatrix} \rho \\ \theta \eta \end{pmatrix},$$

where $s \in V_n \cup I$, $\rho \in (V_t \cup V_n \cup Q \cup I)^*$, and $\theta, \kappa, \eta \in (B \cup L)^*$. Moreover, every affix-variable which occurs in $\theta \kappa$ occurs also in $\theta \eta$. This is a consequence of condition c3 in the definition of a well-formed AG. We call s the left-side head of this meta-rule.

(2) For each symbol ι in I , P_A contains exactly one meta-rule whose left-side head is ι . Meta-rules of this form we call copy-meta-rules.

We call $\rho \in (V_t \cup V_n \cup Q \cup I)^*$ the head string, and $\eta \in (B \cup L)^*$ the tail string, of the pair $\begin{pmatrix} \rho \\ \eta \end{pmatrix}$.

We define production rules in G_A analogously to production rules in G (see section 1.1). That is, if the result of replacing each variable by an affix in its domain, throughout the meta-rule

$$\begin{pmatrix} s \\ \theta \kappa \end{pmatrix} : \begin{pmatrix} \rho \\ \theta \eta \end{pmatrix},$$

is $\begin{pmatrix} s \\ \alpha \beta \end{pmatrix} : \begin{pmatrix} \rho \\ \alpha \mu \end{pmatrix}$, then $\begin{pmatrix} s \\ \alpha \beta \end{pmatrix} \rightarrow \begin{pmatrix} \rho \\ \alpha \mu \end{pmatrix}$ is a production rule of G_A . Or, if q is a primitive predicate symbol, then $\begin{pmatrix} q \\ \alpha \beta \end{pmatrix} \rightarrow \begin{pmatrix} \lambda \\ \alpha \end{pmatrix}$ is a production rule of G_A if and only if $F_q(\alpha, \beta) = \underline{\text{true}}$.

We can make the following observations about the production rules of G_A .

(3) There is a one-to-one correspondence between the production rules of G and those production rules of G_A whose left-side-heads are in $V_n U Q$.

(4) Every production rule in G_A is of the form $\begin{pmatrix} s \\ \alpha\beta \end{pmatrix} \rightarrow \begin{pmatrix} \rho \\ \alpha\mu \end{pmatrix}$, where $s \in V_n U Q U I$, $\rho \in (V_t U V_n U Q U I)^*$, and $\alpha, \beta, \mu \in L^*$.

(5) If $\begin{pmatrix} v \\ \alpha\beta \end{pmatrix} \rightarrow \begin{pmatrix} \rho \iota x \sigma \\ \alpha \mu \zeta \phi \nu \end{pmatrix}$, where $v \in V_n$, $\iota \in I$, $x \in V_n U Q$, $\rho, \sigma \in (V_t U V_n U Q U I)^*$, and $\alpha, \beta, \mu, \zeta, \phi, \nu \in L^*$, and where ζ and ϕ are respectively the inherited and derived affixes of x , then also $\begin{pmatrix} \iota \\ \alpha \mu \zeta \end{pmatrix} \rightarrow \begin{pmatrix} \lambda \\ \alpha \mu \end{pmatrix}$. Production rules of the latter form we call copy-rules.

We define derivations in G_A to be right derivations. If $s \in V_n U Q U I$, $\sigma, \rho \in (V_t U V_n U Q U I)^*$, $\tau \in V_t^*$, and $\alpha, \beta, \mu, \nu \in L^*$, then $\begin{pmatrix} \sigma s \tau \\ \nu \alpha \beta \end{pmatrix} \Rightarrow \begin{pmatrix} \sigma \rho \tau \\ \nu \alpha \mu \end{pmatrix}$ if and only if $\begin{pmatrix} s \\ \alpha \beta \end{pmatrix} \rightarrow \begin{pmatrix} \rho \\ \alpha \mu \end{pmatrix}$. This is a direct derivation in G_A . If $\begin{pmatrix} \rho_0 \\ \mu_0 \end{pmatrix} \Rightarrow \begin{pmatrix} \rho_1 \\ \mu_1 \end{pmatrix} \Rightarrow \dots \Rightarrow \begin{pmatrix} \rho_n \\ \mu_n \end{pmatrix}$, where $n \geq 0$, then $\begin{pmatrix} \rho_0 \\ \mu_0 \end{pmatrix} \Rightarrow^* \begin{pmatrix} \rho_n \\ \mu_n \end{pmatrix}$. This is a derivation in G_A .

The language generated by G_A is

$$\mathcal{L}(G_A) = \left\{ \tau \in V_t^* \mid \begin{pmatrix} e \\ \lambda \end{pmatrix} \Rightarrow^* \begin{pmatrix} \tau \\ \lambda \end{pmatrix} \right\}.$$

An auxiliary grammar of the AG of figure 1.1 is shown in figure 2.4, and a derivation of a sentence in this auxiliary grammar is shown in figure 2.5 (compare figure 1.2).

We now prove, in stages, that G_A is "loosely equivalent" to G , that is, G and G_A generate the same language, and each derivation in G of a sentence is somehow closely related to the derivation in G_A of the same sentence.

Lemma 2.2. To every canonical form ϕ of an affix grammar G there corresponds exactly one sentential form ϕ_A in its auxiliary grammar G_A such that the sequence of production rules applied in a canonical derivation of ϕ corresponds to the sequence of production rules applied in the derivation of ϕ_A , except possibly for intervening applications of copy-rules in the latter derivation. Moreover, if

$I = \{ \iota_1, \iota_2, \iota_3, \iota_4, \iota_5, \iota_6, \iota_7, \iota_8, \iota_9 \}$

$P_A :-$

- (p1) $\left(\text{block} \right)_L : \left(\underline{\text{var}} \text{ declns } \underline{\text{begin}} \iota_1 \text{ stmts } \underline{\text{end}} \right)_L .$
- (p2) $\left(\text{declns} \right)_L : \left(\text{empty} \right)_L .$
- (p3) $\left(\text{declns} \right)_L : \left(\text{declns } \text{tag} \text{ : type } \text{:} \iota_2 \text{ declare} \right)_{L1 T M L1 T M L} .$
- (p4) $\left(\text{stmts} \right)_L : \left(\iota_3 \text{ stmt} \right)_L ;$
- (p5) $\left(\iota_4 \text{ stmts } \text{:} \iota_5 \text{ stmt} \right)_{L L L} .$
- (p6) $\left(\text{stmt} \right)_L : \left(\iota_6 \text{ vble } \text{:} \iota_7 \text{ vble } \iota_8 \text{ equal} \right)_{L L M L M1 M1 M} .$
- (p7) $\left(\text{vble} \right)_{L M} : \left(\text{tag } \iota_9 \text{ identify} \right)_{L T L T M} .$
- (p8) $\left(\text{tag} \right)_T : \left(x \text{ tagx} \right)_T ;$
- (p9) $\left(y \text{ tagy} \right)_T ;$
- (p10) $\left(z \text{ tagz} \right)_T .$
- (p11) $\left(\text{type} \right)_M : \left(\underline{\text{int}} \text{ model} \right)_M ;$
- (p12) $\left(\underline{\text{bool}} \text{ modelb} \right)_M .$
- (p13) $\left(\iota_1 \right)_{L L} : \left(L \right)_L .$
- (p14) $\left(\iota_2 \right)_{L1 T M L1 T M} : \left(L1 T M \right)_L .$
- (p15) $\left(\iota_3 \right)_{L L} : \left(L \right)_L .$
- (p16) $\left(\iota_4 \right)_{L L} : \left(L \right)_L .$

(continued)

Figure 2.4. The full auxiliary grammar of the AG of figure 1.1.

$$(p17) \quad \left(\begin{smallmatrix} \mathfrak{L}^5 \\ L \ L \ L \end{smallmatrix} \right) : \left(\begin{smallmatrix} L \ L \end{smallmatrix} \right) .$$

$$(p18) \quad \left(\begin{smallmatrix} \mathfrak{L}^6 \\ L \ L \end{smallmatrix} \right) : \left(\begin{smallmatrix} L \end{smallmatrix} \right) .$$

$$(p19) \quad \left(\begin{smallmatrix} \mathfrak{L}^7 \\ L \ L \ M \ L \end{smallmatrix} \right) : \left(\begin{smallmatrix} L \ L \ M \end{smallmatrix} \right) .$$

$$(p20) \quad \left(\begin{smallmatrix} \mathfrak{L}^8 \\ L \ L \ M \ L \ M1 \ M1 \ M \end{smallmatrix} \right) : \left(\begin{smallmatrix} L \ L \ M \ L \ M1 \end{smallmatrix} \right) .$$

$$(p21) \quad \left(\begin{smallmatrix} \mathfrak{L}^9 \\ L \ T \ L \ T \end{smallmatrix} \right) : \left(\begin{smallmatrix} L \ T \end{smallmatrix} \right) .$$

Figure 2.4 (concluded)

(block)

$\Rightarrow \left(\frac{\text{var}}{\text{xiyi}} \text{ declns } \frac{\text{begin } \text{11} \text{ stmts } \text{end}}{\text{xiyi}} \right)$
 $\Rightarrow \left(\frac{\text{var}}{\text{xiyi}} \text{ declns } \frac{\text{begin } \text{11} \text{ 13 stmt } \text{end}}{\text{xiyi xiyi}} \right)$
 $\Rightarrow \left(\frac{\text{var}}{\text{xiyi}} \text{ declns } \frac{\text{begin } \text{11} \text{ 13 16 vble := 17 vble 18 equal } \text{end}}{\text{xiyi xiyi xiyi i xiyi i i i}} \right)$
 $\Rightarrow \left(\frac{\text{var}}{\text{xiyi}} \text{ declns } \frac{\text{begin } \text{11} \text{ 13 16 vble := 17 vble 18 } \text{end}}{\text{xiyi xiyi xiyi i xiyi i i i}} \right)$
 $\Rightarrow \left(\frac{\text{var}}{\text{xiyi}} \text{ declns } \frac{\text{begin } \text{11} \text{ 13 16 vble := 17 vble } \text{end}}{\text{xiyi xiyi xiyi i xiyi i}} \right)$
 $\Rightarrow \left(\frac{\text{var}}{\text{xiyi}} \text{ declns } \frac{\text{begin } \text{11} \text{ 13 16 vble := 17 tag 19 identify } \text{end}}{\text{xiyi xiyi xiyi i xiyi x xiyi x i}} \right)$
 $\Rightarrow \left(\frac{\text{var}}{\text{xiyi}} \text{ declns } \frac{\text{begin } \text{11} \text{ 13 16 vble := 17 tag 19 } \text{end}}{\text{xiyi xiyi xiyi i xiyi x xiyi x}} \right)$
 $\Rightarrow \left(\frac{\text{var}}{\text{xiyi}} \text{ declns } \frac{\text{begin } \text{11} \text{ 13 16 vble := 17 tag } \text{end}}{\text{xiyi xiyi xiyi i xiyi x}} \right)$
 $\Rightarrow \left(\frac{\text{var}}{\text{xiyi}} \text{ declns } \frac{\text{begin } \text{11} \text{ 13 16 vble := 17 x tagx } \text{end}}{\text{xiyi xiyi xiyi i xiyi x}} \right)$
 $\Rightarrow \left(\frac{\text{var}}{\text{xiyi}} \text{ declns } \frac{\text{begin } \text{11} \text{ 13 16 vble := 17 x } \text{end}}{\text{xiyi xiyi xiyi i xiyi}} \right)$
 $\Rightarrow \left(\frac{\text{var}}{\text{xiyi}} \text{ declns } \frac{\text{begin } \text{11} \text{ 13 16 vble := x } \text{end}}{\text{xiyi xiyi xiyi i}} \right)$
 $\Rightarrow \left(\frac{\text{var}}{\text{xiyi}} \text{ declns } \frac{\text{begin } \text{11} \text{ 13 16 tag 19 identify := x } \text{end}}{\text{xiyi xiyi xiyi y xiyi y i}} \right)$
 $\Rightarrow \left(\frac{\text{var}}{\text{xiyi}} \text{ declns } \frac{\text{begin } \text{11} \text{ 13 16 tag 19 := x } \text{end}}{\text{xiyi xiyi xiyi y xiyi y}} \right)$
 $\Rightarrow \left(\frac{\text{var}}{\text{xiyi}} \text{ declns } \frac{\text{begin } \text{11} \text{ 13 16 tag := x } \text{end}}{\text{xiyi xiyi xiyi y}} \right)$

(continued)

Figure 2.5 A derivation of a sentence in the grammar of figure 2.4.

$$\Rightarrow \left(\frac{\text{var declns}}{\text{xiyi}} \frac{\text{begin } \iota^1 \iota^3 \iota^6 y \text{ tagy} := x \text{ end}}{\text{xiyi xiyi xiyi xiyi y}} \right)$$

$$\Rightarrow \left(\frac{\text{var declns}}{\text{xiyi}} \frac{\text{begin } \iota^1 \iota^3 \iota^6 y := x \text{ end}}{\text{xiyi xiyi xiyi}} \right)$$

$$\Rightarrow \left(\frac{\text{var declns}}{\text{xiyi}} \frac{\text{begin } \iota^1 \iota^3 y := x \text{ end}}{\text{xiyi xiyi}} \right)$$

$$\Rightarrow \left(\frac{\text{var declns}}{\text{xiyi}} \frac{\text{begin } \iota^1 y := x \text{ end}}{\text{xiyi}} \right)$$

$$\Rightarrow \left(\frac{\text{var declns}}{\text{xiyi}} \text{begin } y := x \text{ end} \right)$$

$$\Rightarrow \left(\frac{\text{var declns}}{\text{xi}} \text{tag} : \text{type} ; \iota^2 \text{declare } \frac{\text{begin } y := x \text{ end}}{\text{xi y i xiyi}} \right)$$

$$\Rightarrow \left(\frac{\text{var declns}}{\text{xi}} \text{tag} : \text{type} ; \iota^2 \frac{\text{begin } y := x \text{ end}}{\text{xi y i}} \right)$$

$$\Rightarrow \left(\frac{\text{var declns}}{\text{xi}} \text{tag} : \text{type} ; \frac{\text{begin } y := x \text{ end}}{\text{i}} \right)$$

$$\Rightarrow \left(\frac{\text{var declns}}{\text{xi}} \text{tag} : \text{int model} ; \frac{\text{begin } y := x \text{ end}}{\text{i}} \right)$$

$$\Rightarrow \left(\frac{\text{var declns}}{\text{xi}} \text{tag} : \text{int} ; \frac{\text{begin } y := x \text{ end}}{\text{y}} \right)$$

$$\Rightarrow \left(\frac{\text{var declns}}{\text{xi}} \text{y tagy} : \text{int} ; \frac{\text{begin } y := x \text{ end}}{\text{y}} \right)$$

$$\Rightarrow \left(\frac{\text{var declns}}{\text{xi}} \text{y} : \text{int} ; \text{begin } y := x \text{ end} \right)$$

$$\Rightarrow \left(\frac{\text{var declns}}{\lambda} \text{tag} : \text{type} ; \iota^2 \text{declare } \frac{\text{y} : \text{int} ; \text{begin } y := x \text{ end}}{\lambda \text{ x i xi}} \right)$$

$$\Rightarrow \left(\frac{\text{var declns}}{\lambda} \text{tag} : \text{type} ; \iota^2 \frac{\text{y} : \text{int} ; \text{begin } y := x \text{ end}}{\lambda \text{ x i}} \right)$$

$$\Rightarrow \left(\frac{\text{var declns}}{\lambda} \text{tag} : \text{type} ; \text{y} : \text{int} ; \text{begin } y := x \text{ end} \right)$$

(continued)

Figure 2.5 (continued)

$$\begin{aligned}
&\Rightarrow \left(\underline{\text{var}} \underset{\lambda}{\text{declns}} \underset{x}{\text{tag}} : \underset{i}{\text{int}} \text{ model } ; y : \text{int} ; \underline{\text{begin}} y := x \underline{\text{end}} \right) \\
&\Rightarrow \left(\underline{\text{var}} \underset{\lambda}{\text{declns}} \underset{x}{\text{tag}} : \text{int} ; y : \text{int} ; \underline{\text{begin}} y := x \underline{\text{end}} \right) \\
&\Rightarrow \left(\underline{\text{var}} \underset{\lambda}{\text{declns}} x \underset{x}{\text{tagx}} : \text{int} ; y : \text{int} ; \underline{\text{begin}} y := x \underline{\text{end}} \right) \\
&\Rightarrow \left(\underline{\text{var}} \underset{\lambda}{\text{declns}} x : \text{int} ; y : \text{int} ; \underline{\text{begin}} y := x \underline{\text{end}} \right) \\
&\Rightarrow \left(\underline{\text{var}} \underset{\lambda}{\text{empty}} x : \text{int} ; y : \text{int} ; \underline{\text{begin}} y := x \underline{\text{end}} \right) \\
&\Rightarrow \left(\underline{\text{var}} x : \text{int} ; y : \text{int} ; \underline{\text{begin}} y := x \underline{\text{end}} \right)
\end{aligned}$$

Figure 2.5 (concluded)

N.B. λ here represents an affix which is the empty string; it does not represent the empty sequence of affixes.

- (i) $\phi = \pi_1 \dots \pi_m \pi \tau$, where $\pi_1, \dots, \pi_m, \pi, \tau$ are as defined in theorem 2.1;
- (ii) for each $i \in [1, m]$, $\begin{pmatrix} x_{i-1} \\ \alpha_{i-1} \beta_{i-1} \end{pmatrix} \rightarrow \begin{pmatrix} \rho_i \iota_i x_i \rho_i' \\ \alpha_{i-1} \mu_i \alpha_i \beta_i \mu_i' \end{pmatrix}$ corresponds to $x_{i-1}(\alpha_{i-1}; \beta_{i-1}) \rightarrow \pi_i x_i(\alpha_i; \beta_i) \pi_i'$, and it is understood that, if $\alpha_i = \lambda$ then $\iota_i = \lambda$, otherwise $\iota_i \in I$;
- and (iii) $\begin{pmatrix} x_m \\ \alpha_m \beta_m \end{pmatrix} \rightarrow \begin{pmatrix} \rho \\ \alpha_m \mu \end{pmatrix}$ corresponds to $x_m(\alpha_m; \beta_m) \rightarrow \pi$;
- then $\phi_A = \begin{pmatrix} \rho_1 \iota_1 \dots \rho_m \iota_m \rho \tau \\ \mu_1 \alpha_1 \dots \mu_m \alpha_m \end{pmatrix}$.

Proof. We prove the theorem by induction on the number of steps in the derivation of ϕ . The postulate is true for derivations of one step, since these must be of the forms $e(\lambda; \lambda) \Rightarrow \pi$ in G and $\begin{pmatrix} e \\ \lambda \end{pmatrix} \Rightarrow \begin{pmatrix} \rho \\ \mu \end{pmatrix}$ in G_A , where $e(\lambda; \lambda) \rightarrow \pi$ and $\begin{pmatrix} e \\ \lambda \end{pmatrix} \rightarrow \begin{pmatrix} \rho \\ \mu \end{pmatrix}$ are corresponding production rules.

Suppose $e(\lambda; \lambda) \Rightarrow^* \phi$ is a derivation of n steps. Suppose further, as in the proof of theorem 2.1, that $\pi_{k+1}, \dots, \pi_m, \pi \in V_t^*$ (where $1 \leq k \leq m$) but that $\pi_k = \pi' x'(\alpha'; \beta') \gamma$. Then, by definition of the auxiliary grammar, $\rho_{k+1} = \pi_{k+1}$, \dots , $\rho_m = \pi_m$, $\rho = \pi$, and $\mu_{k+1} = \dots = \mu_m = \mu = \lambda$; also ρ_k, μ_k must be of the forms $\rho_k = \rho' \iota' x' \gamma$, $\mu_k = \mu' \alpha' \beta'$, where it is understood that, if $\alpha' = \lambda$ then $\iota' = \lambda$, otherwise $\iota' \in I$.

Then the derivation in G_A continues

$$\begin{aligned}
 \phi_A &= \begin{pmatrix} \rho_1 \iota_1 \dots \iota_{m-1} \rho_m \iota_m \pi \tau \\ \mu_1 \alpha_1 \dots \alpha_{m-1} \mu_m \alpha_m \end{pmatrix} \\
 &\Rightarrow^* \begin{pmatrix} \rho_1 \iota_1 \dots \iota_{m-1} \rho_m \pi \tau \\ \mu_1 \alpha_1 \dots \alpha_{m-1} \mu_m \end{pmatrix} \\
 &= \begin{pmatrix} \rho_1 \iota_1 \dots \iota_{m-1} \pi_m \\ \mu_1 \alpha_1 \dots \alpha_{m-1} \end{pmatrix} \\
 &\vdots \\
 &\Rightarrow^* \begin{pmatrix} \rho_1 \iota_1 \dots \iota_{k-1} \rho_k \pi_{k+1} \dots \pi_m \pi \tau \\ \mu_1 \alpha_1 \dots \alpha_{k-1} \mu_k \end{pmatrix} \\
 &= \begin{pmatrix} \rho_1 \iota_1 \dots \iota_{k-1} \rho' \iota' x' \gamma \pi_{k+1} \dots \pi_m \pi \tau \\ \mu_1 \alpha_1 \dots \alpha_{k-1} \mu' \alpha' \beta' \end{pmatrix} \\
 &\Rightarrow \begin{pmatrix} \rho_1 \iota_1 \dots \iota_{k-1} \rho' \iota' \rho'' \gamma \pi_{k+1} \dots \pi_m \pi \tau \\ \mu_1 \alpha_1 \dots \alpha_{k-1} \mu' \alpha' \mu'' \end{pmatrix} = \phi_A' .
 \end{aligned}$$

At each step of this derivation, except the last, there are two possibilities ($k \leq i \leq m$):-

(1) $\iota_i = \alpha_i = \lambda$. Then

$$\left(\begin{array}{c} \rho_1 \iota_1 \dots \rho_i \iota_i \pi_{i+1} \dots \pi_m \pi \tau \\ \mu_1 \alpha_1 \dots \mu_i \alpha_i \end{array} \right) = \left(\begin{array}{c} \rho_1 \iota_1 \dots \rho_i \pi_{i+1} \dots \pi_m \pi \tau \\ \mu_1 \alpha_1 \dots \mu_i \end{array} \right) .$$

(2) $\iota_i \in I$. Then there is a production rule

$$\left(\begin{array}{c} \iota_i \\ \alpha_{i-1} \mu_i \alpha_i \end{array} \right) \rightarrow \left(\begin{array}{c} \lambda \\ \alpha_{i-1} \mu_i \end{array} \right) ,$$

and therefore

$$\left(\begin{array}{c} \rho_1 \iota_1 \dots \iota_{i-1} \rho_i \iota_i \pi_{i+1} \dots \pi_m \pi \tau \\ \mu_1 \alpha_1 \dots \alpha_{i-1} \iota_i \alpha_i \end{array} \right) \Rightarrow \left(\begin{array}{c} \rho_1 \iota_1 \dots \iota_{i-1} \rho_i \pi_{i+1} \dots \pi_m \pi \tau \\ \mu_1 \alpha_1 \dots \alpha_{i-1} \mu_i \end{array} \right) .$$

In either case the relation \Rightarrow^* holds. (Notice that a production rule whose left-side head is ι_i is always applicable when ι_i is due to be replaced in a sentential form, and that there is exactly one applicable production rule.)

The last step of the derivation is an application of the production rule $\left(\begin{array}{c} x' \\ \alpha' \beta' \end{array} \right) \rightarrow \left(\begin{array}{c} \rho'' \\ \alpha' \mu'' \end{array} \right)$ which corresponds to $x'(\alpha'; \beta') \rightarrow \pi''$. (Note that the first of these is applicable if and only if the second is applicable.)

Now, the $(n+1)$ -th step in the derivation in G , from the proof of theorem 2.1, was

$$\phi \Rightarrow \pi_1 \dots \pi_{k-1} \pi' \pi'' \gamma \pi_{k+1} \dots \pi_m \pi \tau = \phi' .$$

ϕ_A' bears the same relationship to ϕ' as ϕ_A did to ϕ . The derivation $\phi_A \Rightarrow^* \phi_A'$ consisted of some applications of copy-rules followed by an application of the production rule in G_A which corresponds to the production rule applied in the direct derivation $\phi \Rightarrow \phi'$. Moreover, exactly the same choice of production rules was available in each derivation.

Similar logic holds in the case $\pi \notin V_t^*$. Thus the inductive hypothesis holds for derivations of $(n+1)$ steps in G , and the theorem is proved.

Lemma 2.3. An affix grammar G and its auxiliary grammar G_A generate the same language.

Proof. Consider the general canonical form of G (from theorem 2.1),

$$\phi = \pi_1 \dots \pi_m \pi \tau,$$

and the corresponding sentential form in G_A (from lemma 2.2),

$$\phi_A = \begin{pmatrix} \rho_1 \iota_1 \dots \rho_m \iota_m \rho \tau \\ \mu_1 \alpha_1 \dots \mu_m \alpha_m \mu \end{pmatrix}.$$

If ϕ is a sentence, then $\pi_1, \dots, \pi_m, \pi \in V_t^*$, whence it follows, by definition of an auxiliary grammar, that $\rho_1 = \pi_1, \dots, \rho_m = \pi_m, \rho = \pi$, and $\mu_1 = \mu_2 = \dots = \mu_m = \mu = \lambda$. Then there is a unique derivation from ϕ_A as follows:-

$$\begin{aligned} \phi_A &= \begin{pmatrix} \rho_1 \iota_1 \dots \iota_{m-1} \rho_m \iota_m \pi \tau \\ \mu_1 \alpha_1 \dots \alpha_{m-1} \mu_m \alpha_m \end{pmatrix} \\ &\Rightarrow^* \begin{pmatrix} \rho_1 \iota_1 \dots \iota_{m-1} \rho_m \pi \tau \\ \mu_1 \alpha_1 \dots \alpha_{m-1} \mu_m \end{pmatrix} \\ &= \begin{pmatrix} \rho_1 \iota_1 \dots \iota_{m-1} \pi_m \pi \tau \\ \mu_1 \alpha_1 \dots \alpha_{m-1} \end{pmatrix} \\ &\vdots \\ &\Rightarrow^* \begin{pmatrix} \rho_1 \iota_1 \rho_2 \pi_3 \dots \pi_m \pi \tau \\ \mu_1 \alpha_1 \mu_2 \end{pmatrix} \\ &= \begin{pmatrix} \rho_1 \iota_1 \pi_2 \pi_3 \dots \pi_m \pi \tau \\ \mu_1 \alpha_1 \end{pmatrix} \\ &\Rightarrow^* \begin{pmatrix} \rho_1 \pi_2 \pi_3 \dots \pi_m \pi \tau \\ \mu_1 \end{pmatrix} \\ &= \begin{pmatrix} \pi_1 \dots \pi_m \pi \tau \\ \lambda \end{pmatrix}. \end{aligned}$$

(See the justification, up to before its last step, of the derivation $\phi_A \Rightarrow^* \phi_A'$ in the proof of lemma 2.2. The last step in the present derivation is possible because

$$\begin{pmatrix} \rho \\ \lambda \end{pmatrix} \rightarrow \begin{pmatrix} \rho_1 \iota_1 x_1 \rho_1' \\ \mu_1 \alpha_1 \beta_1 \mu_1' \end{pmatrix} \text{ implies } \begin{pmatrix} \iota_1 \\ \mu_1 \alpha_1 \end{pmatrix} \rightarrow \begin{pmatrix} \lambda \\ \mu_1 \end{pmatrix}, \text{ unless } \alpha_1 = \lambda.)$$

Thus, if ϕ is a sentence, then ϕ_A derives that same sentence and no other. Since there is a one-to-one correspondence

between forms such as ϕ and ϕ_A , there must also be a one-to-one correspondence between the sentences of G and those of G_A . Thus the lemma is proved.

Theorem 2.4. An affix grammar G and its auxiliary grammar G_A are "loosely equivalent", that is, they generate the same language and there is a one-to-one correspondence between the canonical derivations of sentences in G and the derivations of the same sentences in G_A .

Proof. This theorem follows from a combination of the proofs of lemmas 2.2 and 2.3.

2.4. Optimisations to an Auxiliary Grammar

Our objective in defining an auxiliary grammar in section 2.3 was to find a grammar which, while preserving the essential structure of the affix grammar, would reflect better than the AG the transformations taking place during a left-to-right parse. For this purpose the essential properties of the auxiliary grammar are (1) its loose equivalence to the AG; (2) its special treatment of inherited affixes, as reflected in the copy-rules; and (3) the fact that every affix-variable occurring on the left side of a meta-rule occurs also on its right side.

In general, an auxiliary grammar which satisfies these requirements for a given AG is not unique. An auxiliary grammar constructed by the method given in section 2.3 can be subjected to various optimisations which reduce the number of copy-meta-rules and shorten the other meta-rules. These optimisations were not introduced in section 2.2 in order not to complicate the proof of the equivalence of the auxiliary grammar and the original AG; however, we show informally that none of the optimisations invalidate the proof.

The first optimisation concerns the role of the copy-symbols. It often happens in practice that, in a meta-rule

$$\begin{pmatrix} v \\ \xi\eta \end{pmatrix} : \begin{pmatrix} \rho l x \rho' \\ \xi v \theta \kappa v' \end{pmatrix} ,$$

where θ and κ respectively occupied the inherited and derived affix-positions of x in the original AG meta-rule, θ is a suffix of ξv , and therefore, in every production rule $\begin{pmatrix} v \\ \alpha\beta \end{pmatrix} \rightarrow \begin{pmatrix} \rho l x \rho' \\ \alpha\mu \xi \phi \mu \end{pmatrix}$ which can be generated from the meta-rule, ξ is a suffix of $\alpha\mu$. Consider what happens to ξ during a derivation after an application of this production rule. x must eventually be replaced by application of a production rule $\begin{pmatrix} x \\ \xi\phi \end{pmatrix} \rightarrow \begin{pmatrix} \sigma \\ \xi\psi \end{pmatrix}$. Thus ξ is undisturbed, and remains undisturbed in the sentential form until x has been completely replaced by a terminal string. Then the production rule $\begin{pmatrix} l \\ \alpha\mu\xi \end{pmatrix} \rightarrow \begin{pmatrix} \lambda \\ \alpha\mu \end{pmatrix}$ is applied to remove both l and ξ . All this time, however, $\alpha\mu$ has been situated in the tail string immediately to the left of ξ , and ξ is a copy of a suffix of $\alpha\mu$. Thus it is not necessary for ξ to be separately present in the tail string; the suffix of $\alpha\mu$ will do the job of ξ perfectly satisfactorily. We can achieve this effect by replacing the meta-rule quoted above by

$$\begin{pmatrix} v \\ \xi\eta \end{pmatrix} : \begin{pmatrix} \rho x \rho' \\ \xi v \kappa v' \end{pmatrix}$$

and discarding the meta-rule which has l as its left-side head. (And, since l no longer occurs in any meta-rule, l may be removed from I .) If a copy-symbol l' occurs in ρ' , then the meta-rule whose left-side head is l' must be correspondingly altered, by the removal of θ from both sides, so that observation (5) in section 2.3 will continue to be true.

It may be observed that we have here a generalisation of the case $\theta=\lambda$, which we considered in section 2.3. In lemma 2.2, for example, we generalise the phrase "it is understood that, if $\alpha_i=\lambda$ then $l_i=\lambda$, otherwise $l_i \in I$ " to "it is understood that both l_i and α_i may stand for λ , provided that α_i is guaranteed to be a suffix of $\alpha_{i-1}\mu_i$, otherwise $l_i \in I$ ". Now, in the proof of this lemma we must consider the possibility that $\rho_k=\rho'x'\gamma$ and $\mu_k=\mu'\beta'$, where α' is a suffix of $\alpha_{k-1}\mu'$, say $\alpha_{k-1}\mu'=\alpha'\alpha'$. In this case the last step of

the derivation in G_A is

$$\begin{aligned}
 & \left(\begin{array}{c} \rho_1 \iota_1 \dots \rho_{k-1} \iota_{k-1} \rho_k \pi_{k+1} \dots \pi_m \pi \tau \\ \mu_1 \alpha_1 \dots \mu_{k-1} \alpha_{k-1} \mu_k \end{array} \right) \\
 &= \left(\begin{array}{c} \rho_1 \iota_1 \dots \rho_{k-1} \iota_{k-1} \rho' x' \gamma \pi_{k+1} \dots \pi_m \pi \tau \\ \mu_1 \alpha_1 \dots \mu_{k-1} \psi \alpha' \beta' \end{array} \right) \\
 &\Rightarrow \left(\begin{array}{c} \rho_1 \iota_1 \dots \rho_{k-1} \iota_{k-1} \rho' \rho'' \gamma \pi_{k+1} \dots \pi_m \pi \tau \\ \mu_1 \alpha_1 \dots \mu_{k-1} \psi \alpha' \mu'' \end{array} \right) \\
 &= \left(\begin{array}{c} \rho_1 \iota_1 \dots \rho_{k-1} \iota_{k-1} \rho' \rho'' \gamma \pi_{k+1} \dots \pi_m \pi \tau \\ \mu_1 \alpha_1 \dots \mu_{k-1} \alpha_{k-1} \mu' \mu'' \end{array} \right)
 \end{aligned}$$

The final sentential form satisfies the modified inductive hypothesis.

The auxiliary grammar of figure 2.4 illustrates well the practical value of this optimisation. For example, ι_1 can be eliminated because 'L' is a suffix of 'L' and ι_2 because 'L1 T M' is a suffix of 'L1 T M'. In fact every symbol in I can be eliminated with the exceptions of ι_7 and ι_8 . As a result of the removal of ι_6 from the right side of meta-rule p6, meta-rules p19 and p20 must be altered to

$$\left(\begin{array}{c} \iota_7 \\ L \ M \ L \end{array} \right) : \left(\begin{array}{c} \lambda \\ L \ M \end{array} \right) \quad \text{and} \quad \left(\begin{array}{c} \iota_8 \\ L \ M \ L \ M_1 \ M_1 \ M \end{array} \right) : \left(\begin{array}{c} \lambda \\ L \ M \ L \ M_1 \end{array} \right)$$

respectively.

The second optimisation exploits the fact, noted in the proof of lemma 2.2, that, whenever a copy-symbol ι is due to be replaced in a sentential form, a production rule generated from the meta-rule whose left-side head is ι is always applicable. If the meta-rule is

$$\left(\begin{array}{c} \iota \\ \xi \eta \end{array} \right) : \left(\begin{array}{c} \lambda \\ \xi \end{array} \right) ,$$

and if ξ has a prefix θ such that $\xi = \theta \kappa$ and every affix-variable occurring in η also occurs in κ , then we can simplify the meta-rule to

$$\left(\begin{array}{c} \iota \\ \kappa \eta \end{array} \right) : \left(\begin{array}{c} \lambda \\ \kappa \end{array} \right) .$$

The correctness of this optimisation can be proved along similar lines to the previous proof.

In the auxiliary grammar of figure 2.4, after the first optimisation the prefix 'L' can be cancelled on the left and right sides of meta-rule p20. The resulting optimised auxiliary grammar is shown in figure 2.6.

The third optimisation enables copy-meta-rules which are "similar" to one another to be replaced by a single meta-rule. Two copy-meta-rules are similar if their left-side tail strings could be made identical by a systematic renaming of the variables occurring in one of them, provided that such a renaming preserves the domains of the variables. Similar copy-meta-rules generate sets of production rules which differ only in their left-side heads. One of the copy-meta-rules can be discarded, and the symbol which was its left-side head can be replaced (in the right side of the meta-rule where it occurs) by the symbol which is the head of the other copy-meta-rule.

For example, the meta-rules

$$\begin{pmatrix} \iota \\ M & L & M \end{pmatrix} : \begin{pmatrix} \lambda \\ M & L \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} \iota' \\ M1 & L & M1 \end{pmatrix} : \begin{pmatrix} \lambda \\ M1 & L \end{pmatrix}$$

are similar, as can be seen by replacing 'M1' by 'M' throughout the second meta-rule, provided that 'M1' and 'M' have the same associated affix-nonterminal. If these meta-rules occurred in an auxiliary grammar, the second could be eliminated and ι' replaced by ι in the meta-rule in whose right side ι' occurs.

Figure 2.7 shows a derivation in the optimised auxiliary grammar of figure 2.6. This is shorter than the derivation of the same sentence in the unoptimised auxiliary grammar (figure 2.5), but retains a loose equivalence to the derivation in the affix grammar (figure 1.2). The shortening of the derivation will, of course, be mirrored by a shortening of the parse of the sentence, and this is a consequence of our first optimisation.

The second optimisation has no independent practical significance, but it tends to increase the number of meta-rules to which the third optimisation may be applied. The importance of the

$I = \{ \iota 7, \iota 8 \}$

$P_A :-$

- (p1) $\left(\text{block} \right)_L : \left(\underline{\text{var}} \text{ declns } \underline{\text{begin}} \text{ stmts } \underline{\text{end}} \right) .$
- (p2) $\left(\text{declns} \right)_L : \left(\text{empty} \right)_L .$
- (p3) $\left(\text{declns} \right)_L : \left(\text{declns}_{L1} \text{ tag}_T \text{ ; type}_M \text{ ; declare}_L \right) .$
- (p4) $\left(\text{stmts} \right)_L : \left(\text{stmt} \right)_L ;$
- (p5) $\left(\text{stmts} \right)_L : \left(\text{stmts} \text{ ; stmt} \right)_L .$
- (p6) $\left(\text{stmt} \right)_L : \left(\text{vble}_{LM} := \iota 7 \text{ vble}_{LM1} \iota 8 \text{ equal}_{M1M} \right) .$
- (p7) $\left(\text{vble} \right)_{LM} : \left(\text{tag}_{LT} \text{ identify}_M \right) .$
- (p8) $\left(\text{tag} \right)_T : \left(x \text{ tagx}_T \right) ;$
- (p9) $\left(y \text{ tagy}_T \right) ;$
- (p10) $\left(z \text{ tagz}_T \right) .$
- (p11) $\left(\text{type} \right)_M : \left(\underline{\text{int}} \text{ model}_M \right) ;$
- (p12) $\left(\underline{\text{bool}} \text{ modeb}_M \right) .$
- (p19) $\left(\iota 7_{LM L} \right) : \left(L M \right) .$
- (p20) $\left(\iota 8_{MLM1M1M} \right) : \left(M L M1 \right) .$

Figure 2.6. An optimised auxiliary grammar of the AG of figure 1.1.

$$\begin{aligned}
&\Rightarrow \left(\frac{\text{var}}{x1} \text{ declns } \frac{\text{tag}}{y} : \frac{\text{type}}{i} ; \text{ declare } \frac{\text{begin}}{xiyi} y := x \text{ end} \right) \\
&\Rightarrow \left(\frac{\text{var}}{x1} \text{ declns } \frac{\text{tag}}{y} : \frac{\text{type}}{i} ; \text{ begin } y := x \text{ end} \right) \\
&\Rightarrow \left(\frac{\text{var}}{x1} \text{ declns } \frac{\text{tag}}{y} : \frac{\text{int}}{i} \text{ mode1} ; \text{ begin } y := x \text{ end} \right) \\
&\Rightarrow \left(\frac{\text{var}}{x1} \text{ declns } \frac{\text{tag}}{y} : \text{int} ; \text{ begin } y := x \text{ end} \right) \\
&\Rightarrow \left(\frac{\text{var}}{x1} \text{ declns } y \text{ tagy} : \text{int} ; \text{ begin } y := x \text{ end} \right) \\
&\Rightarrow \left(\frac{\text{var}}{x1} \text{ declns } y : \text{int} ; \text{ begin } y := x \text{ end} \right) \\
&\Rightarrow \left(\frac{\text{var}}{\lambda} \text{ declns } \frac{\text{tag}}{x} : \frac{\text{type}}{i} ; \text{ declare } y : \frac{\text{int}}{xi} ; \text{ begin } y := x \text{ end} \right) \\
&\Rightarrow \left(\frac{\text{var}}{\lambda} \text{ declns } \frac{\text{tag}}{x} : \frac{\text{type}}{i} ; y : \text{int} ; \text{ begin } y := x \text{ end} \right) \\
&\Rightarrow \left(\frac{\text{var}}{\lambda} \text{ declns } \frac{\text{tag}}{x} : \frac{\text{int}}{i} \text{ mode1} ; y : \text{int} ; \text{ begin } y := x \text{ end} \right) \\
&\Rightarrow \left(\frac{\text{var}}{\lambda} \text{ declns } \frac{\text{tag}}{x} : \text{int} ; y : \text{int} ; \text{ begin } y := x \text{ end} \right) \\
&\Rightarrow \left(\frac{\text{var}}{\lambda} \text{ declns } x \text{ tagx} : \text{int} ; y : \text{int} ; \text{ begin } y := x \text{ end} \right) \\
&\Rightarrow \left(\frac{\text{var}}{\lambda} \text{ declns } x : \text{int} ; y : \text{int} ; \text{ begin } y := x \text{ end} \right) \\
&\Rightarrow \left(\frac{\text{var}}{\lambda} \text{ empty } x : \text{int} ; y : \text{int} ; \text{ begin } y := x \text{ end} \right) \\
&\Rightarrow \left(\text{var } x : \text{int} ; y : \text{int} ; \text{ begin } y := x \text{ end} \right)
\end{aligned}$$

Figure 2.7 (concluded)

latter is that it will make our left-to-right parsing method more likely to work on the grammar. The reason for this is that merging of similar copy-meta-rules reduces the probability of our parser's having a state in which it knows that it must make a reduction involving a copy-rule, but not which one. The first optimisation is also helpful in this respect, and in addition it reduces the probability of our parser's having a state in which it cannot resolve even whether a reduction involving a copy-rule must be made.

2.5. Head Grammar of an Auxiliary Grammar; AF-LR(k) Grammars

To summarise the results of sections 2.3 and 2.4, we have shown how to construct from an AG G an auxiliary grammar G_A which is loosely equivalent to G . We can solve the problem of parsing sentences in G by solving the problem of parsing sentences in G_A . Each meta-rule of G_A is of the form

$$\begin{pmatrix} s \\ \kappa \end{pmatrix} : \begin{pmatrix} \rho \\ \theta \end{pmatrix} ,$$

where $s \in (V_n \cup I)$, $\rho \in (V_t \cup V_n \cup Q \cup I)^*$, $\theta, \kappa \in (B \cup L)^*$, and every affix-variable occurring in κ occurs also in θ , if G is well-formed.

In this section no real distinction will be made between members of V_n and of I , and this will simplify our progress.

Let $\{\#_0, \#_1, \dots\}$ be a set of special symbols, called #-symbols, disjoint from $V_t \cup V_n \cup Q \cup I$, such that each #-symbol is uniquely associated with either a meta-rule in P_A or with a primitive predicate symbol in Q .

It follows from this definition that each production rule in G_A is associated with exactly one #-symbol: if the left-side head of the production rule is in $V_n \cup I$, then it is the #-symbol associated with the meta-rule from which the production rule was generated; if the left-side head is a primitive predicate symbol q , then it is the #-symbol associated with q . Of course, the

reverse is not true: there may be an unbounded number of production rules associated with a given #-symbol.

Suppose $\begin{pmatrix} s \\ \emptyset \end{pmatrix} \rightarrow \begin{pmatrix} \rho \\ \mu \end{pmatrix}$ is a production rule whose associated #-symbol is $\#_j$. If there is a derivation in G_A

$$\begin{pmatrix} e \\ \lambda \end{pmatrix} \Rightarrow^* \begin{pmatrix} \sigma s \tau \\ \nu \emptyset \end{pmatrix} \Rightarrow \begin{pmatrix} \sigma \rho \tau \\ \nu \mu \end{pmatrix} ,$$

then $\sigma \rho \#_j$ is a characteristic head string of the sentential form $\begin{pmatrix} \sigma \rho \tau \\ \nu \mu \end{pmatrix}$.

Given a characteristic head string $\sigma \#_j$ of $\begin{pmatrix} \sigma \tau \\ \nu \end{pmatrix}$, the production rule applied in the last step of a derivation of $\begin{pmatrix} \sigma \tau \\ \nu \end{pmatrix}$ can be re-constructed as follows.

(1) If $\#_j$ is associated with a meta-rule

$$\begin{pmatrix} s \\ \kappa \end{pmatrix} : \begin{pmatrix} \rho \\ \theta \end{pmatrix} ,$$

then the suffix μ which has the same length as θ is detached from ν . Each affix-variable occurring in θ is given the value of the corresponding affix in μ . Since every affix-variable occurring in κ occurs also in θ , the tail string \emptyset corresponding to κ can now be constructed. The applied production rule is then $\begin{pmatrix} s \\ \emptyset \end{pmatrix} \rightarrow \begin{pmatrix} \rho \\ \mu \end{pmatrix}$.

(2) If $\#_j$ is associated with a primitive predicate symbol q , then the suffix μ whose length equals the number of inherited affix-positions of q is detached from ν . The derived affixes are now determined by $\beta = \tilde{F}_q(\mu)$. The applied production rule is then $\begin{pmatrix} q \\ \mu \beta \end{pmatrix} \rightarrow \begin{pmatrix} \lambda \\ \mu \end{pmatrix}$.

Finally, the sentential form which existed immediately before the application of this production rule can be re-constructed by reversing the direct derivation.

We now go on to obtain some results about the set of characteristic strings of our auxiliary grammar.

Theorem 2.5. Every characteristic head string of an auxiliary grammar $G_A = (V_n, V_t, A_n, A_t, Q, I, e, R, B, D, S, P_A)$ is a sentence of the CFG $G' = (V_T', V_N', e', P')$, where

$$\begin{aligned} V_T' &= V_t \cup V_n \cup Q \cup I \cup \{\#_0, \#_1, \dots\} \\ V_N' &= \{ s' \mid s \in V_n \cup Q \cup I \} \\ P' &= \{ s' \rightarrow \rho \#_j \mid \begin{pmatrix} s \\ \kappa \end{pmatrix} : \begin{pmatrix} \rho \\ \theta \end{pmatrix} \text{ is in } P_A \text{ and is associated with } \#_j \} \\ &\quad \cup \{ q' \rightarrow \#_j \mid q \text{ is in } Q \text{ and is associated with } \#_j \} \\ &\quad \cup \{ s' \rightarrow \rho x' \mid \begin{pmatrix} s \\ \kappa \end{pmatrix} : \begin{pmatrix} \rho x \sigma \\ \theta \end{pmatrix} \text{ is in } P_A, \text{ and } x \in V_n \cup Q \cup I \} \end{aligned}$$

Proof. Observe that P' may equivalently be defined by

$$\begin{aligned} P' &= \{ s' \rightarrow \rho \#_j \mid \begin{pmatrix} s \\ \emptyset \end{pmatrix} \rightarrow \begin{pmatrix} \rho \\ \mu \end{pmatrix} \text{ is a production rule in } G_A \\ &\quad \text{and is associated with } \#_j \} \\ &\quad \cup \{ s' \rightarrow \rho x' \mid \begin{pmatrix} s \\ \emptyset \end{pmatrix} \rightarrow \begin{pmatrix} \rho x \sigma \\ \mu \end{pmatrix} \text{ is a production rule in } G_A, \\ &\quad \text{and } x \in V_n \cup Q \cup I \} \end{aligned}$$

Every sentential form in G_A is of the form $\begin{pmatrix} \rho_1 \dots \rho_m \rho_T \\ \mu_1 \dots \mu_m \mu \end{pmatrix}$, where $\rho_1, \dots, \rho_m, \rho \in (V_t \cup V_n \cup Q \cup I)^*$ and $T \in V_t^*$ and $\mu_1, \dots, \mu_m, \mu \in L^*$, where for each $i \in [1, m]$ there is a production rule $\begin{pmatrix} s_{i-1} \\ \emptyset_{i-1} \end{pmatrix} \rightarrow \begin{pmatrix} \rho_i s_i \rho_i' \\ \mu_i \emptyset_i \mu_i' \end{pmatrix}$, where $s_0 = e$ and $\emptyset_0 = \lambda$, and where there is a production rule $\begin{pmatrix} s_m \\ \emptyset_m \end{pmatrix} \rightarrow \begin{pmatrix} \rho \\ \mu \end{pmatrix}$. This

may be proved, by induction on the number of steps in the derivation, in a manner similar to the proof of theorem 2.1.

A characteristic head string of $\begin{pmatrix} \rho_1 \dots \rho_m \rho_T \\ \mu_1 \dots \mu_m \mu \end{pmatrix}$ is $\rho_1 \dots \rho_m \rho \#_j$, where $\#_j$ is the $\#$ -symbol associated with the production rule $\begin{pmatrix} s_m \\ \emptyset_m \end{pmatrix} \rightarrow \begin{pmatrix} \rho \\ \mu \end{pmatrix}$. This characteristic head string is generated by the sequence

$$\begin{aligned} e' &\Rightarrow \rho_1 s_1' && (\text{applying } e' \rightarrow \rho_1 s_1') \\ &\Rightarrow \rho_1 \rho_2 s_2' && (\text{applying } s_1' \rightarrow \rho_2 s_2') \\ &\vdots && \\ &\Rightarrow \rho_1 \dots \rho_m s_m' && (\text{applying } s_{m-1}' \rightarrow \rho_m s_m') \\ &\Rightarrow \rho_1 \dots \rho_m \rho \#_j && (\text{applying } s_m' \rightarrow \rho \#_j) \end{aligned}$$

which is a derivation in G' . Thus the theorem is proved.

We now define the head grammar of the auxiliary grammar G_A to be the CFG $G_H = (V_t, V_h, e, P_h)$, where

$$\begin{aligned} V_h &= V_n U Q U I, \\ P_h &= \left\{ s \rightarrow \rho \mid \begin{pmatrix} s \\ \kappa \end{pmatrix} : \begin{pmatrix} \rho \\ \theta \end{pmatrix} \text{ is in } P_A \right\} \\ &\quad \cup \left\{ q \rightarrow \lambda \mid q \in Q \right\}. \end{aligned}$$

As P_h contains exactly one production rule for each meta-rule in P_A and each primitive predicate symbol in Q , there is a one-to-one correspondence between the members of P_h and our set of #-symbols.

As an example, figure 2.8 shows the head grammar of the auxiliary grammar of figure 2.6.

Theorem 2.6. Every characteristic head string of an auxiliary grammar G_A is a characteristic string of G_A 's head grammar G_H .

Proof. From section 2.1, the characteristic grammar of G_H , i.e. the grammar which generates exactly the set of characteristic strings of G_H , is (V'_t, V'_n, e, P') , where

$$\begin{aligned} V'_t &= V_t U V_h U \{ \#_0, \#_1, \dots \} \\ &= V_t U V_n U Q U I U \{ \#_0, \#_1, \dots \}, \\ V'_n &= \{ s' \mid s \in V_h \} \\ &= \{ s' \mid s \in V_n U Q U I \}, \\ P' &= \left\{ s' \rightarrow \rho \#_j \mid s \rightarrow \rho \text{ is in } P_h \text{ and is associated with } \#_j \right\} \\ &\quad \cup \left\{ s' \rightarrow \rho x' \mid s \rightarrow \rho x \sigma \text{ is in } P_h, \text{ and } x \in V_h \right\} \\ &= \left\{ s' \rightarrow \rho \#_j \mid \begin{pmatrix} s \\ \kappa \end{pmatrix} : \begin{pmatrix} \rho \\ \theta \end{pmatrix} \text{ is in } P_A \text{ and is associated with } \#_j \right\} \\ &\quad \cup \left\{ q' \rightarrow \#_j \mid q \text{ is in } Q \text{ and is associated with } \#_j \right\} \\ &\quad \cup \left\{ s' \rightarrow \rho x' \mid \begin{pmatrix} s \\ \kappa \end{pmatrix} : \begin{pmatrix} \rho x \sigma \\ \theta \end{pmatrix} \text{ is in } P_A, \text{ and } x \in V_n U Q U I \right\}. \end{aligned}$$

This is precisely the grammar G' of theorem 2.5, which generates all the characteristic head strings of G_A . Thus the theorem is proved.

The foregoing results are of the greatest importance, as they establish a firm connection between AGs and CFGs through auxiliary

$$V_h = \{ \text{block, declns, stmts, stmt, tag, type,} \\ \text{empty, declare, equal, identify,} \\ \text{tagx, tagy, tagz, mode1, modeb, } \iota_7, \iota_8 \}$$

P_h :-

- (p1) block -> var declns begin stmts end
- (p2) declns -> empty
- (p3) declns -> declns tag : type ; declare
- (p4) stmts -> stmt
- (p5) stmts -> stmts ; stmt
- (p6) stmt -> vble := ι_7 vble ι_8 equal
- (p7) vble -> tag identify
- (p8) tag -> x tagx
- (p9) tag -> y tagy
- (p10) tag -> z tagz
- (p11) type -> int mode1
- (p12) type -> bool modeb
- (p19) ι_7 ->
- (p20) ι_8 ->
- (p22) empty ->
- (p23) declare ->
- (p24) equal ->
- (p25) identify ->
- (p26) tagx ->
- (p27) tagy ->
- (p28) tagz ->
- (p29) mode1 ->
- (p30) modeb ->

Figure 2.8 Head grammar of the auxiliary grammar of figure 2.6.

grammars and head grammars of the former. This prompts the following definitions.

An auxiliary grammar is AF-LR(k) (for "affix-free LR(k)") if and only if its head grammar is LR(k). Likewise, an auxiliary grammar is AF-LALR(k) (respectively, AF-SLR(k)) if and only if its head grammar is LALR(k) (respectively, SLR(k)).

If an auxiliary grammar is AF-LR(k), then every sentential form $\begin{pmatrix} \sigma\tau \\ \nu \end{pmatrix}$, except $\begin{pmatrix} e \\ \lambda \end{pmatrix}$, has a unique characteristic head string $\sigma\#_j$ which can be determined by inspecting only σ and the first k terminals of τ , i.e. without regard to the tail string ν . This is the reason for our terminology "affix-free".

We are now in a position to construct parsers for AF-LR(k) auxiliary grammars. This we do in section 2.7, after first considering the special case of AF-LR(0) grammars in section 2.6.

2.6. Parsers for AF-LR(0) Grammars

By the definition at the end of section 2.5, an auxiliary grammar G_A is AF-LR(0) if and only if its head grammar G_H is LR(0), that is if G_H 's CFSM contains no inadequate states.

Our parser for G_A will be a generalisation of the LR(0) parser described in section 2.1. It is based on G_H 's CFSM, and uses two stacks - a state stack, as in the LR(0) parser, and an affix stack, on which is stored the tail string of the current (alleged) sentential form. The parsing algorithm is as follows.

- Step 1. Start the CFSM in its initial state. Initialise both stacks to be empty. Be prepared to read the first terminal of the input string to be parsed.
- Step 2. Stack the name of the current state on the state stack. If the current state is a read state, go to step 3. If the current state is a reduce state, consider the #-transition out of this state, and go to step 4.
- Step 3. Read the next terminal from the input string. If there is a transition out of the current state under that terminal, then change to the state at the end of that transition and go to step 2. Otherwise, the input string is not a sentence of G_A .
- Step 4. Determine from the #-symbol on the considered transition, using the method described near the beginning of section 2.5, the production rule $\left(\begin{smallmatrix} s \\ \emptyset \end{smallmatrix}\right) \rightarrow \left(\begin{smallmatrix} p \\ \mu \end{smallmatrix}\right)$ which must have been applied in the last step of the derivation of the current sentential form; μ will be the sequence of affixes occupying the topmost positions in the affix stack. Replace μ by \emptyset on the affix stack. If n is the length of p , pop n state names from the state stack. If $s=e$, the parse has been successfully completed. Otherwise, there will be a transition under s out of the state whose name is

now at the top of the state stack. Change to the state at the end of this transition and go to step 2.

The essential difference between our parsing algorithm and the LR(0) parsing algorithm is in the reduction procedure (step 4). In the LR(0) parser there was no way for the reduction to fail. In our parser, there are two conceivable ways in which the reduction might fail.

(1) If the $\#$ -symbol is associated with a meta-rule

$$\begin{pmatrix} s \\ \mu \end{pmatrix} : \begin{pmatrix} p \\ \theta \end{pmatrix} ,$$

then the only possibility of failure is if μ cannot be matched to θ , that is if either (i) an affix-variable occurs more than once in θ and the corresponding affixes in μ are not all equal, or (ii) an affix occurs in θ but does not equal the corresponding affix in μ . In fact neither of these eventualities can ever arise. As a formal proof of this would be tedious, we illustrate (i) by example. Suppose an AG contains the meta-rules

$$\begin{aligned} f(A; \lambda) &: g(\lambda; B) h(A; \lambda) , \\ h(A; \lambda) &: t . \end{aligned}$$

(We have written these meta-rules in the notation introduced in section 2.2.) The auxiliary grammar will contain the meta-rules

$$\begin{aligned} \begin{pmatrix} f \\ A \end{pmatrix} &: \begin{pmatrix} g \ \iota \ h \\ A \ B \ A \end{pmatrix} , \\ \begin{pmatrix} h \\ A \end{pmatrix} &: \begin{pmatrix} t \\ A \end{pmatrix} , \\ \begin{pmatrix} \iota \\ A \ B \ A \end{pmatrix} &; \begin{pmatrix} \\ A \ B \end{pmatrix} . \end{aligned}$$

Suppose now that we are attempting to reduce $\Phi = \begin{pmatrix} \sigma & g & \iota & h & \tau \\ \nu & c & b & a & \end{pmatrix}$, where $a, c \in L(D(A))$ and $b \in L(D(B))$, using the first meta-rule. If $c \neq a$ then $\mu = 'c \ b \ a'$ cannot be matched to $\theta = 'A \ B \ A'$. But let us re-trace the last step of the parse:

$$\begin{pmatrix} \sigma & g & \iota & h & \tau \\ \nu & c & b & a & \end{pmatrix} \Rightarrow \begin{pmatrix} \sigma & g & \iota & t & \tau \\ \nu & c & b & a & \end{pmatrix} \quad \left(\text{applying } \begin{pmatrix} h \\ a \end{pmatrix} \rightarrow \begin{pmatrix} t \\ a \end{pmatrix} \right).$$

No further derivation is possible, since there is no production rule $\begin{pmatrix} \iota \\ c \ b \ a \end{pmatrix} \rightarrow \begin{pmatrix} \\ c \ b \end{pmatrix}$ if $c \neq a$. Thus Φ could not have been obtained from a terminal string by successive reductions, unless $c=a$. Note that we did not assume that Φ was a sentential form, so our argument is

valid whether the input string is a sentence or not. A similar example could be used to illustrate (ii).

(2) If the $\#$ -symbol is associated with a primitive predicate symbol q , then the reduction will indeed fail if $\tilde{F}_q(\mu) = \omega$. Then the current form $\begin{pmatrix} \sigma\tau \\ \nu\mu \end{pmatrix}$ is not in fact a sentential form. For, if it were, there must be a derivation

$$\begin{pmatrix} e \\ \lambda \end{pmatrix} \Rightarrow^* \begin{pmatrix} \sigma q \tau \\ \nu \mu \beta \end{pmatrix} \Rightarrow \begin{pmatrix} \sigma \tau \\ \nu \mu \end{pmatrix} ,$$

in which case $F_q(\mu, \beta)$ must be true and therefore $\tilde{F}_q(\mu) = \beta$. Thus if a reduction fails, the input string is not a sentence of G_A .

We see here what will happen if the head grammar G_H has a characteristic string $\sigma\#_j$ (where $\#_j$ is associated with a primitive predicate symbol q) which is not also a characteristic head string of G_A . That is, for every sentential form $\begin{pmatrix} \sigma q \tau \\ \nu \mu \beta \end{pmatrix}$, $F_q(\mu, \beta) = \text{false}$. Then the reduce state accessed by σ may indeed be accessed during a parse, but the subsequent reduction will always fail. Such a situation would arise in practice only if a mistake had been made in the design of the original AG; it would indicate that some meta-rule in the AG could not be used in the derivation of any sentence.

We now prove that our parser will always find the correct reduction to make to a sentential form. Consider a sentential form whose derivation in G_A is

$$\begin{pmatrix} e \\ \lambda \end{pmatrix} \Rightarrow^* \begin{pmatrix} \sigma s \tau \\ \nu \emptyset \end{pmatrix} \Rightarrow \begin{pmatrix} \sigma \rho \tau \\ \nu \mu \end{pmatrix} .$$

If $\#_j$ is the $\#$ -symbol associated with the production rule $\begin{pmatrix} s \\ \emptyset \end{pmatrix} \rightarrow \begin{pmatrix} \rho \\ \mu \end{pmatrix}$, then by definition $\sigma\rho\#_j$ is the characteristic head string of $\begin{pmatrix} \sigma \rho \tau \\ \nu \mu \end{pmatrix}$. By theorem 2.6, $\sigma\rho\#_j$ is also a characteristic string of G_H , therefore $\sigma\rho\#_j$ is accepted by G_H 's CFSM. It follows that $\sigma\rho$ accesses a state with a transition under $\#_j$; and, since G_H is LR(0), this state must be a reduce state. Since there is a production rule $\begin{pmatrix} s \\ \emptyset \end{pmatrix} \rightarrow \begin{pmatrix} \rho \\ \mu \end{pmatrix}$, the reduction must work, and the effect of step 4 of our parsing algorithm is to replace ρ by s and μ by \emptyset in the sentential form, thus yielding a new sentential form $\begin{pmatrix} \sigma s \tau \\ \nu \emptyset \end{pmatrix}$.

The CFSM of G_H has special features, when viewed as the basis of our parser. If $\#_j$ is the symbol on the $\#$ -transition out of a reduce state, and if $\#_j$ is associated with a primitive predicate symbol, then we re-define the state to be a predicate state, or if $\#_j$ is associated with a copy-meta-rule, then we re-define the state to be a copy state.

We call a transition under a primitive predicate symbol a predicate-transition, and a transition under a copy-symbol a copy-transition.

Consider a predicate state, N , with $\#_j$ associated with a primitive predicate symbol q . If σ accesses N , then $\sigma\#_j$ is a characteristic string of G_H . Thus, for some $\tau \in V_t^*$, $e \Rightarrow^* \sigma q \tau \Rightarrow \sigma \tau$ is a derivation in G_H , and σq must be a prefix of at least one characteristic string (that of $\sigma q \tau$). Since the CFSM is deterministic, there must be a transition under q from N to another state, say M .

Whenever the parser reaches state N and successfully re-constructs the production rule $\left(\begin{smallmatrix} q \\ \mu \end{smallmatrix} \beta \right) \rightarrow \left(\begin{smallmatrix} \lambda \\ \mu \end{smallmatrix} \right)$ to be involved in the reduction, it will pop 0 state names off the state stack, leaving N still on top. On completing the reduction, therefore, it will traverse the transition under q out of N . Thus the parser in state N will always change to state M . Therefore, since q and $\#_j$ are uniquely associated with each other, the transition under $\#_j$ is redundant and may be eliminated, provided that we modify our parsing algorithm accordingly.

A similar transformation is possible in a copy state with $\#_j$ associated with a copy-symbol ι , since there is exactly one meta-rule in G_A with ι as its left-side head and therefore ι and $\#_j$ are uniquely associated with each other, and since the right-side head string of this meta-rule is empty.

Thus every $\#$ -transition out of a predicate state or a copy state can be eliminated. The only remaining $\#$ -transitions are

those under #-symbols associated with meta-rules of G_A whose left-side heads are nonterminals, that is those meta-rules of G_A which correspond to the meta-rules of the AG from which G_A was constructed. The resulting machine we call the affix finite-state machine (AFSM) of G_A .

Figure 2.9 shows the CFSM of the head grammar of figure 2.8, and figure 2.10 shows the result of applying the above transformation to this CFSM, namely the AFSM of the auxiliary grammar of figure 2.6.

Our parsing algorithm for G_A may now be re-stated in terms of G_A 's AFSM. As before, the algorithm uses a state stack and an affix stack. We call it the AF-LR(0) parsing algorithm.

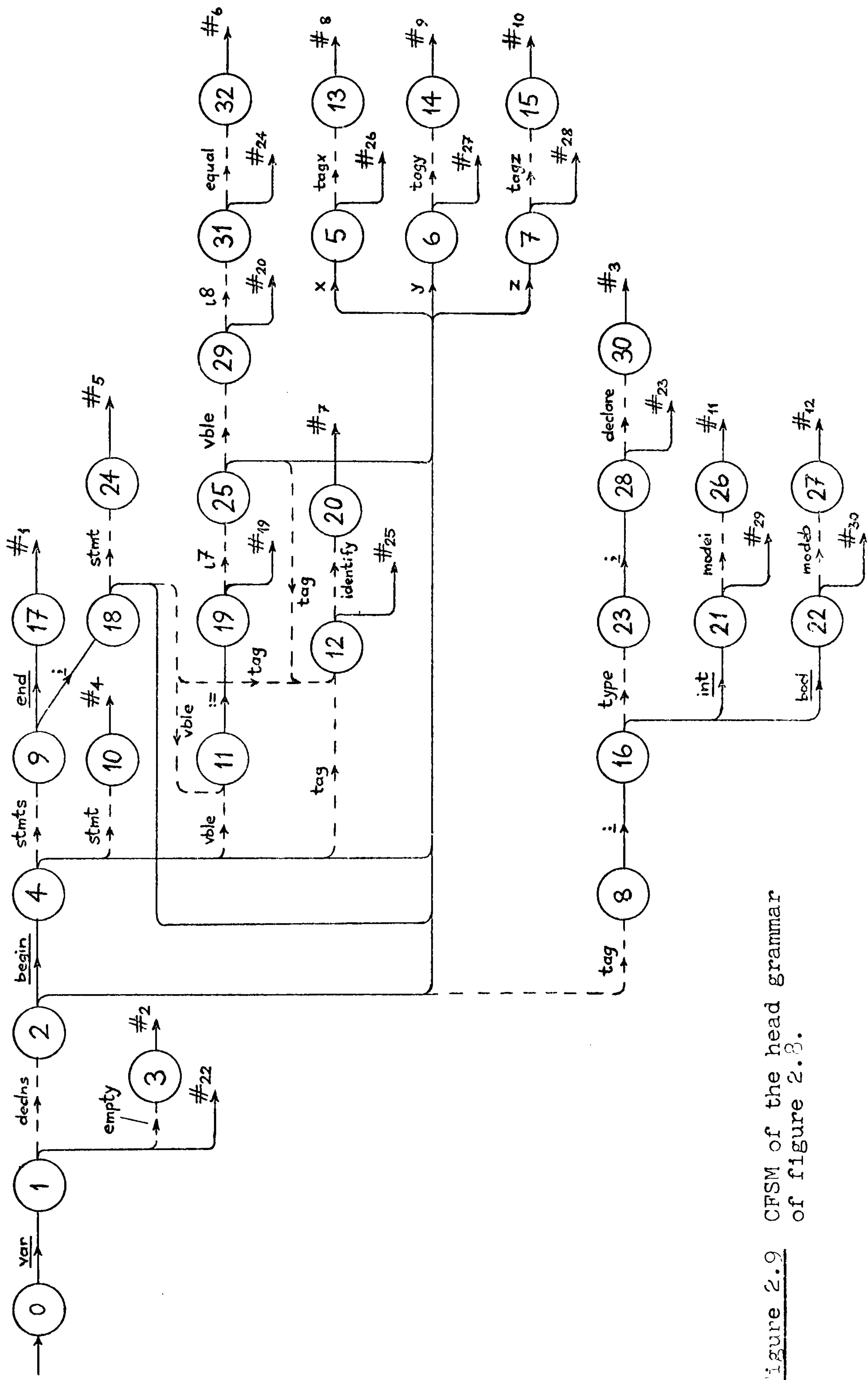
- Step 1. Start the AFSM in its initial state. Initialise both stacks to be empty. Be prepared to read the first terminal of the input string to be parsed.
- Step 2. Stack the name of the current state on the state stack. If the current state is a read state, go to step 3; if a reduce state, consider the #-transition out of this state and go to step 4; if a predicate state, consider the predicate-transition out of this state and go to step 5; if a copy state, consider the copy-transition out of this state and go to step 6.
- Step 3. Read the next terminal from the input string. If there is a transition out of the current state under that terminal, then change to the state at the end of that transition and go to step 2. Otherwise, the input string is not a sentence of G_A .
- Step 4. Let the meta-rule associated with the #-symbol on the considered transition be $\left(\begin{smallmatrix} v \\ k \end{smallmatrix} \right) : \left(\begin{smallmatrix} \rho \\ \theta \end{smallmatrix} \right)$, and let n be the length of ρ , m the length of θ , and p the length of k . Match θ to the top m affixes in the affix stack, and give each affix-variable in θ the value of the corresponding

affix in the stack. Replace the top m affixes in the stack by the p affixes which are the values of the variables of K . Pop n state names from the state stack. If $v=e$, the parse has been successfully completed. Otherwise, there will be a transition under v out of the state whose name is now at the top of the state stack. Change to the state at the end of this transition and go to state 2.

Step 5. Let q be the primitive predicate symbol on the considered transition, and let m and n be the number of inherited and derived affix-positions of q . Apply the function \tilde{F}_q to the top m affixes in the affix stack. If the result is ω , the input string is not a sentence of G_A . Otherwise, stack the n derived affixes yielded by the function on the affix stack. Change to the state at the end of the considered transition and go to step 2.

Step 6. Let the meta-rule whose left-side head is the copy-symbol on the considered transition be $\left(\begin{smallmatrix} l \\ \theta \eta \end{smallmatrix} \right) : \left(\begin{smallmatrix} \lambda \\ \theta \end{smallmatrix} \right)$, and let m be the length of θ and p the length of η . Match θ to the top m affixes in the affix stack, and give each affix-variable in θ the value of the corresponding affix in the stack. Stack the p affixes which are the values of the variables of, or which themselves occur in, η . Change to the state at the end of the considered transition and go to step 2.

Figure 2.11 shows a history of the AF-LR(0) parser based on the AFSM of figure 2.10. (Compare this with figures 1.2 and 2.7.) As an example of the action of the parser in a predicate state, consider the second time it reaches state 28. The function $\tilde{F}_{\text{declare}}$ when applied to the affixes 'xi', 'y' and 'i' yields the affix 'xiyi', which is stacked on top of the first three. As an example of a reduction, consider its subsequent action in state 30, where the meta-rule involved is p_3 (figure 2.6). 'Ll', 'T', 'E' and 'L' are matched to 'xi', 'y', 'i' and 'xiyi' respectively; the left-side tail string consists simply of the affix matched to 'L', namely 'xiyi', which replaces the top 4 affixes in the stack.



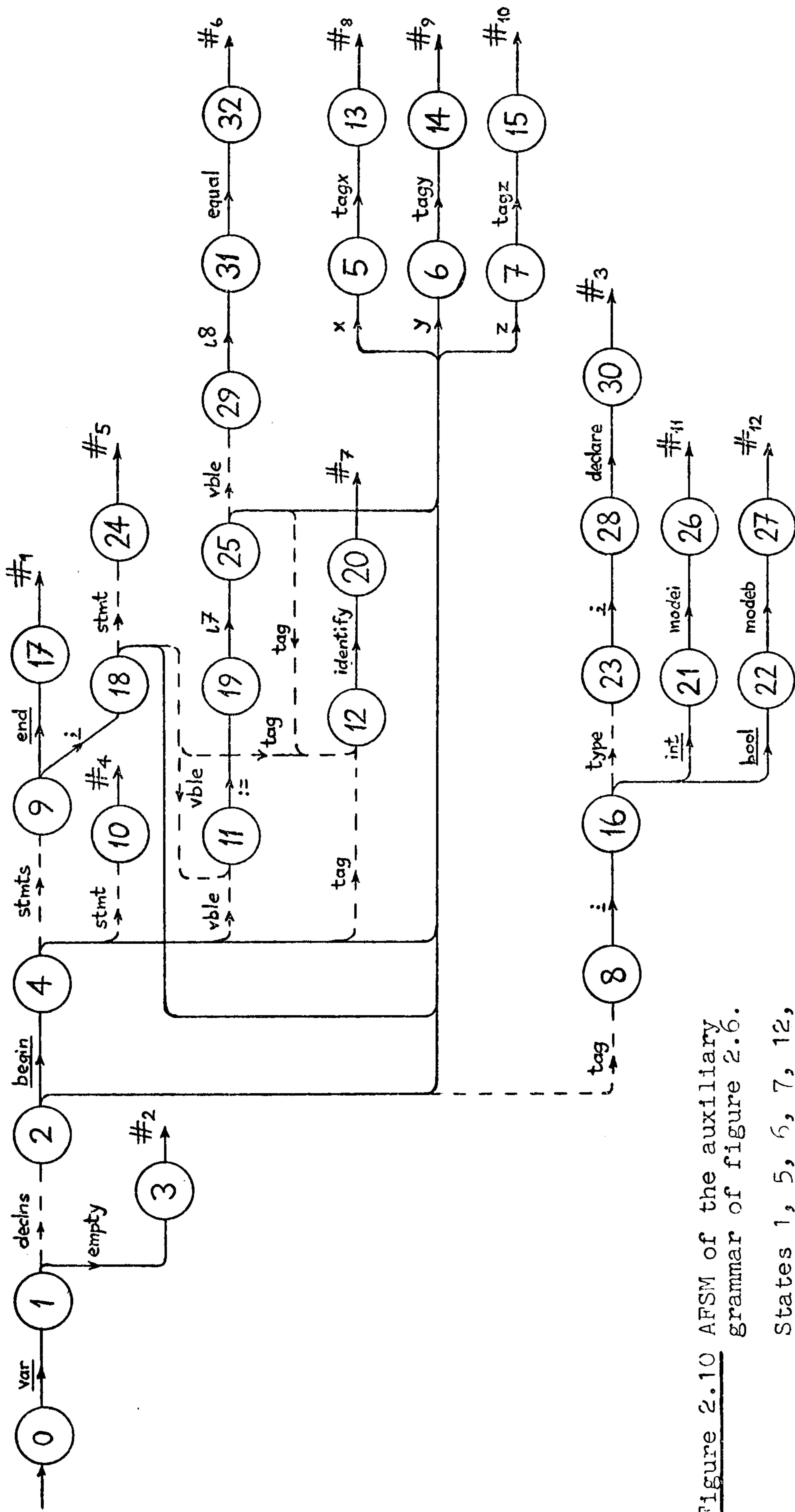


Figure 2.10 AFSM of the auxiliary grammar of figure 2.6.

States 1, 5, 6, 7, 12, 21, 22, 28, 31 are predicate states; states 19 and 29 are copy states.

State	State stack	Affix stack	Remaining input string	Reduction
0			<u>var</u> x : <u>int</u> ...	
1	0		x : <u>int</u> ; ...	
3	0 1		x : <u>int</u> ; ...	p2
2	0 1		x : <u>int</u> ; ...	
5	0 1 2		: <u>int</u> ; ...	
13	0 1 2 5	λ x	: <u>int</u> ; ...	p8
8	0 1 2	λ x	: <u>int</u> ; ...	
16	0 1 2 8	λ x	<u>int</u> ; ...	
21	0 1 2 8 16	λ x	; y : <u>int</u> ...	
26	0 1 2 8 16 21	λ x i	; y : <u>int</u> ...	p11
23	0 1 2 8 16	λ x i	; y : <u>int</u> ...	
28	0 1 2 8 16 23	λ x i	y : <u>int</u> ; ...	
30	0 1 2 8 16 23 28	λ x i xi	y : <u>int</u> ; ...	p3
2	0 1	xi	y : <u>int</u> ; ...	
6	0 1 2	xi	: <u>int</u> ; ...	
14	0 1 2 6	xi y	: <u>int</u> ; ...	p9
8	0 1 2	xi y	: <u>int</u> ; ...	
16	0 1 2 8	xi y	<u>int</u> ; ...	
21	0 1 2 8 16	xi y	; <u>begin</u> ...	
26	0 1 2 8 16 21	xi y i	; <u>begin</u> ...	p11
23	0 1 2 8 16	xi y i	; <u>begin</u> ...	
28	0 1 2 8 16 23	xi y i	<u>begin</u> y := x ...	
30	0 1 2 8 16 23 28	xi y i xiyi	<u>begin</u> y := x ...	p3

(continued)

Figure 2.11 History of the AF-LR(0) parser of the auxiliary grammar of figure 2.6 when applied to the sentence var x : int ; y : int ; begin y := x end .

State	State stack	Affix stack	Remaining input string	Reduction
2	0 1	xiyi	<u>begin</u> y := x ...	
4	0 1 2	xiyi	y := x <u>end</u>	
6	0 1 2 4	xiyi	:= x <u>end</u>	
14	0 1 2 4 6	xiyi y	:= x <u>end</u>	p9
12	0 1 2 4	xiyi y	:= x <u>end</u>	
20	0 1 2 4 12	xiyi y i	:= x <u>end</u>	p7
11	0 1 2 4	xiyi i	:= x <u>end</u>	
19	0 1 2 4 11	xiyi i	x <u>end</u>	
25	0 1 2 4 11 19	xiyi i xiyi	x <u>end</u>	
5	0 1 2 4 11 19 25	xiyi i xiyi	<u>end</u>	
13	0 1 2 4 11 19 25 5	xiyi i xiyi x	<u>end</u>	p8
12	0 1 2 4 11 19 25	xiyi i xiyi x	<u>end</u>	
20	0 1 2 4 11 19 25 12	xiyi i xiyi x i	<u>end</u>	p7
29	0 1 2 4 11 19 25	xiyi i xiyi i	<u>end</u>	
31	0 1 2 4 11 19 25 29	xiyi i xiyi i i i	<u>end</u>	
32	0 1 2 4 11 19 25 29 31	xiyi i xiyi i i i	<u>end</u>	p6
10	0 1 2 4	xiyi	<u>end</u>	p4
9	0 1 2 4	xiyi	<u>end</u>	
17	0 1 2 4 9	xiyi		p1

Figure 2.11 (concluded)

(Only the first few symbols of the remaining input string are shown on each line.)

Note that, when we speak of a "reduction" in the context of our AF-LR(0) parsing algorithm (and, later, our AF-LR(k) parsing algorithm), we are referring to the reversed application of a production rule whose left-side head is a nonterminal.

Theorem 2.7. Let G_A be an AF-LR(0) auxiliary grammar of an AG G . Then the sequence of production rules involved in reductions and (implicitly) in traversals of predicate-transitions when the AF-LR(0) parsing algorithm is applied to a sentence of G corresponds exactly to the parse of that sentence in G .

Proof. By lemma 2.3, G and G_A generate the same language.

Consider the parse of the sentence in G_A . We showed that this parse is correctly determined by the parsing algorithm using the CFSM of G_A 's head grammar. By construction of G_A 's AFSM from this CFSM, this parse is also determined by the AF-LR(0) parsing algorithm if production rules implicitly involved in traversals of predicate- and copy-transitions are recorded as well as reductions. The sequence mentioned in the statement of this theorem is this parse with copy-rules deleted.

By lemma 2.2, the parse of the sentence in G_A corresponds to the parse of the sentence in G except for the insertion of copy-rules. Thus the theorem is proved.

The significance of this result is that, although it may be possible to construct a number of different auxiliary grammars, and hence a number of different AFSMs, from a given AG, every one of these AFSMs yields the same output when applied to a given sentence, and in that sense they are equivalent to one another and are characterised by the original AG.

2.7. Parsers for AF-LR(k) Grammars

We have defined an auxiliary grammar G_A to be AF-LR(k) if and only if its head grammar G_H is LR(k). The techniques discussed in section 2.1 can be immediately applied to G_H 's CFSM to compute the k-symbol look-ahead sets associated with transitions out of inadequate states and, if necessary, to split states.

The observations in the previous section about predicate-transitions out of predicate states apply equally to predicate-transitions out of look-ahead states. Predicate-transitions and transitions under the corresponding #-symbols always occur in pairs. However, the k-symbol look-ahead set associated with each #-transition out of a k-look-ahead state must be retained and associated with the corresponding predicate-transition before the #-transition is eliminated.

All this, of course, applies to copy-transitions out of look-ahead states.

We can now generalise the AF-LR(0) parsing algorithm of the previous section to the AF-LR(k) parsing algorithm. Step 2 is modified by the addition of the sentence "If the current state is a k-look-ahead state, go to step 7.". A new step is added, as follows.

Step 7. Examine the next k terminals of the input string, but do not read them. If this k-terminal string is not in any of the k-symbol look-ahead sets associated with transitions out of the current state, then the input string is not a sentence of G_A . Otherwise, the string must be in exactly one of these look-ahead sets. If the transition associated with this set is a terminal-transition, read a terminal from the input string, change to the state at the end of the transition, and go to step 2. If the transition is a #-transition, a predicate-transition, or a copy-transition, consider this transition and go to step 4, step 5, or step

6, respectively.

Theorem 2.7 can be extended to prove the correctness of this AF-LR(k) parsing algorithm. If reductions and traversals of predicate-transitions are recorded, the output of the parser corresponds exactly to the parse of the sentence in the AG from which G_A was constructed.

Just as many CFGs of interest are LALR(1), so also we expect many programming languages to be definable by AGs whose (optimised) auxiliary grammars are AF-LALR(1). This would not be surprising, as in a conventional translator the CFG defining the context-free part of the syntax is often used to drive a syntax analyser in which are embedded "semantic" functions which, among other things, check the context-sensitivities. These functions are somewhat analogous to the primitive predicate functions of the AG. The AF-LR(k) condition corresponds to the requirement that the "semantic" functions can be invoked when required, not necessarily when a reduction is being performed.

2.8. Left Recursion in Affix Grammars

Direct left recursion in a CFG, that is a production rule of the form $N \rightarrow Nv$, and indirect left recursion, that is a cycle of production rules $N_1 \rightarrow N_2v_2$, ..., $N_{n-1} \rightarrow N_nv_n$, $N_n \rightarrow N_1v_1$ (where $n > 1$), are well-known problems in respect of top-down parsing methods. Bottom-up parsing methods have no such difficulty as they do not work on the goal-seeking principle. Even the LR(k) method, which is a mixed case with an element of goal-orientation, handles such constructs with ease. For example, figures 2.2(b) and 2.9 show CFSMs of CFGs with left-recursive production rules.

Left recursion can, however, appear in a more subtle form, involving production rules with empty right sides. Consider, for example, the following CFG, which generates the language $\{ ba^n a \mid n \geq 0 \}$:

$S \rightarrow L\lambda$
 $L \rightarrow EL\alpha$
 $L \rightarrow b$
 $E \rightarrow \lambda$

This grammar is not LR(k) for any k, since, in parsing 'baⁿλ', λ must be reduced to 'E' n times before reading 'b', and the necessary decisions cannot be taken without scanning arbitrarily far to the right.

Such "delayed" left recursion is not very likely to arise in ordinary CFGs of interest, but it is highly relevant in the case of a head grammar, which may contain a large number of production rules with empty right sides, namely those with copy-symbols or primitive predicate symbols on their left sides.

Consider, for example, an AG containing the following meta-rule:

$$v(\theta;\kappa) : v(\xi;\eta) \ v \ .$$

The auxiliary grammar will contain meta-rules of the form

$$\begin{pmatrix} v \\ \theta\kappa \end{pmatrix} : \begin{pmatrix} \iota v \rho \\ \theta\xi\eta\mu \end{pmatrix} ,$$

$$\begin{pmatrix} \iota \\ \theta\xi \end{pmatrix} : \begin{pmatrix} \lambda \\ \theta \end{pmatrix} ,$$

and the head grammar will contain production rules of the form

$$v \rightarrow \iota v \rho ,$$

$$\iota \rightarrow \lambda .$$

Thus the head grammar is not LR(k), and therefore the auxiliary grammar is not AF-LR(k). But now suppose that $\xi=\theta$; then we can apply the first optimisation of section 2.4 to eliminate ι , giving the meta-rule

$$\begin{pmatrix} v \\ \theta\kappa \end{pmatrix} : \begin{pmatrix} v \rho \\ \theta\eta\mu \end{pmatrix} ,$$

and now the head grammar contains the direct left-recursive production rule

$$v \rightarrow v \rho ,$$

which we have seen causes no trouble. This example demonstrates

the importance of our first optimisation to auxiliary grammars: it permits the AF-LR(k) parsing method to be applied to an AG containing left-recursive meta-rules, provided that each inherited affix-position of the nonterminal contains the same affix-variable on both sides of each such meta-rule. This seems to happen quite frequently in practice; see for example meta-rule p5 in the AG of figure 1.1.

Delayed left recursion can occur more explicitly in a meta-rule of an AG, if the first nonterminal hypernotation on the right side is preceded by one or more primitive predicate hypernotations. For example, the meta-rule

$$v(\theta;\kappa) : q(\theta;\xi) v(\xi;\eta)$$

will give rise to the following head grammar production rules

$$\begin{aligned} v &\rightarrow qvp & , \\ q &\rightarrow \lambda & . \end{aligned}$$

Thus an AG with explicitly delayed left-recursive meta-rules can never have an AF-LR(k) auxiliary grammar.

Exceptionally, however, such an AG may be transformed by re-ordering delayed left-recursive meta-rules to make them direct left-recursive. Such re-ordering is always subject to condition c3 in the definition of a well-formed AG (section 1.2). For example, the meta-rule

$$v(\theta;\kappa) : q(\xi;\psi) v(\xi;\eta)$$

may be transformed into

$$v(\theta;\kappa) : v(\xi;\eta) q(\xi;\psi) ,$$

provided that no affix-variable has a defining occurrence in ψ and an applied occurrence in ξ (and provided that q 's associated function has no harmful side-effects). If now $\xi=\theta$, then the meta-rule satisfies the condition stated above. Curiously, if $\xi=\theta$, and if the original AG was well-formed, then no variable in ψ can occur also in ξ , so the re-ordering is then subject only to the lack of side-effects in q 's associated function. All this applies also when more than one primitive predicate hypernotation precedes

the nonterminal hypernotation.

We may draw these informal observations together in the following statement: an affix grammar containing left-recursive meta-rules may have an AF-LR(k) auxiliary grammar only if, in every left-recursive meta-rule, each inherited affix-position of the relevant nonterminal hypernotation on the right side contains the same affix-variable as the corresponding affix-position of the left-side hypernotation. This in turn implies that all the nonterminals in an indirect left-recursive cycle have inherited affix-positions with identical domains.

2.9. A-LR(k) Grammars; Multi-Predicate States

Following the lines of our definition of AF-LR(k) grammars, we can define an auxiliary grammar to be A-LR(k) (for "affix LR(k)") if and only if every sentential form $\begin{pmatrix} \sigma\tau \\ \nu \end{pmatrix}$, except $\begin{pmatrix} e \\ \lambda \end{pmatrix}$, has a unique characteristic head string $\sigma\#_j$ which can be determined by inspecting only σ , ν , and the first k terminals of τ .

We do not have any general results about A-LR(k) grammars, but we look at a particular case which is of practical interest. That is when an auxiliary grammar has an AFSM in which every inadequate state which is not k-look-ahead is a multi-predicate state. A multi-predicate state is one out of which there are two or more predicate transitions, but no terminal-, copy- or #-transitions.

Let $\#_1, \dots, \#_n$ be the #-symbols associated with the primitive predicate symbols on the transitions out of a multi-predicate state. If $\sigma \in (V_t \cup V_n \cup Q \cup I)^*$ is any string which accesses this state, then $\sigma\#_1, \dots, \sigma\#_n$ are all characteristic strings of the head grammar. Suppose, however, that, given any $\tau \in V_t^*$ and $\nu \in L^*$ such that $\begin{pmatrix} \sigma\tau \\ \nu \end{pmatrix}$ is a sentential form, at most one of the

primitive predicate functions, when applied to the appropriate suffix of ν , will not yield the result ω ; that is, at most one of $\sigma\#_1, \dots, \sigma\#_n$ is a characteristic head string of (\mathcal{G}^τ) . Then the multi-predicate state is deterministic.

We can give a sufficient, but not necessary, condition for a multi-predicate state to be deterministic. We say that two primitive predicate symbols p and q are disjoint if and only if

- (1) the functions \tilde{F}_p and \tilde{F}_q have the same domains;
and (2) $\{ \alpha \mid \tilde{F}_p(\alpha) \neq \omega \} \cap \{ \alpha \mid \tilde{F}_q(\alpha) \neq \omega \} = \Phi$,

where Φ denotes the empty set. A multi-predicate state is deterministic if the symbols on all the predicate-transitions out of the state are mutually disjoint.

Consider the AG fragment shown in figure 2.12(a). The AFSM of an auxiliary grammar of this AG will contain a multi-predicate state, as shown in figure 2.12(b). It is readily demonstrated that this state satisfies the above condition and is therefore deterministic.

We have found that multi-predicate states satisfying the above condition occur quite commonly in practice. A simple modification to step 5 of the AF-LR(0) (or AF-LR(k)) parsing algorithm suffices to implement such states. If the predicate-transitions out of a multi-predicate state are under q_1, \dots, q_n , then the associated functions $\tilde{F}_{q_1}, \dots, \tilde{F}_{q_n}$ are applied in turn to the affixes at the top of the affix stack, until one of them works (i.e. does not yield ω); then the transition under the corresponding primitive predicate symbol is traversed. Only if all the functions fail (i.e. yield ω) does the parse as a whole fail.

We can now extend the class of left-recursive constructs which we can handle. The AG fragment of figure 2.13(a), for example, contains a delayed left-recursive meta-rule which causes its auxiliary grammar to be non-AF-LR(k). The corresponding part of the AFSM contains two multi-predicate states which are not k -look-ahead but are deterministic. The delayed left-recursive

construct gives rise to a complete loop of predicate- and copy-transitions in the AFSM. Care must therefore be taken to ensure that the parsing algorithm can never loop indefinitely (when the input string is not a sentence). This must be determined by inspection in individual cases; in the grammar of figure 2.13 there is no danger of indefinite looping.

Our method of handling multi-predicate states can be combined with k-symbol look-aheads to resolve more complicated cases of inadequacy. We do not pursue this possibility, as such cases are likely to occur infrequently in practice, and the ideas are more important than the details.

Control:-

v	2	$\{t, t\}$	$\{T, T\}$	
equal	2	$\{t, t\}$	$\{T, T\}$	$\lambda x \lambda y \{x=y\}$
unequal	2	$\{t, t\}$	$\{T, T\}$	$\lambda x \lambda y \{x \neq y\}$

Meta-rules:-

(p1) $v(T1, T2) : \text{equal}(T1, T2) s ;$

(p2) $\text{unequal}(T1, T2) t .$

Figure 2.12 (a) A fragment of an AG.

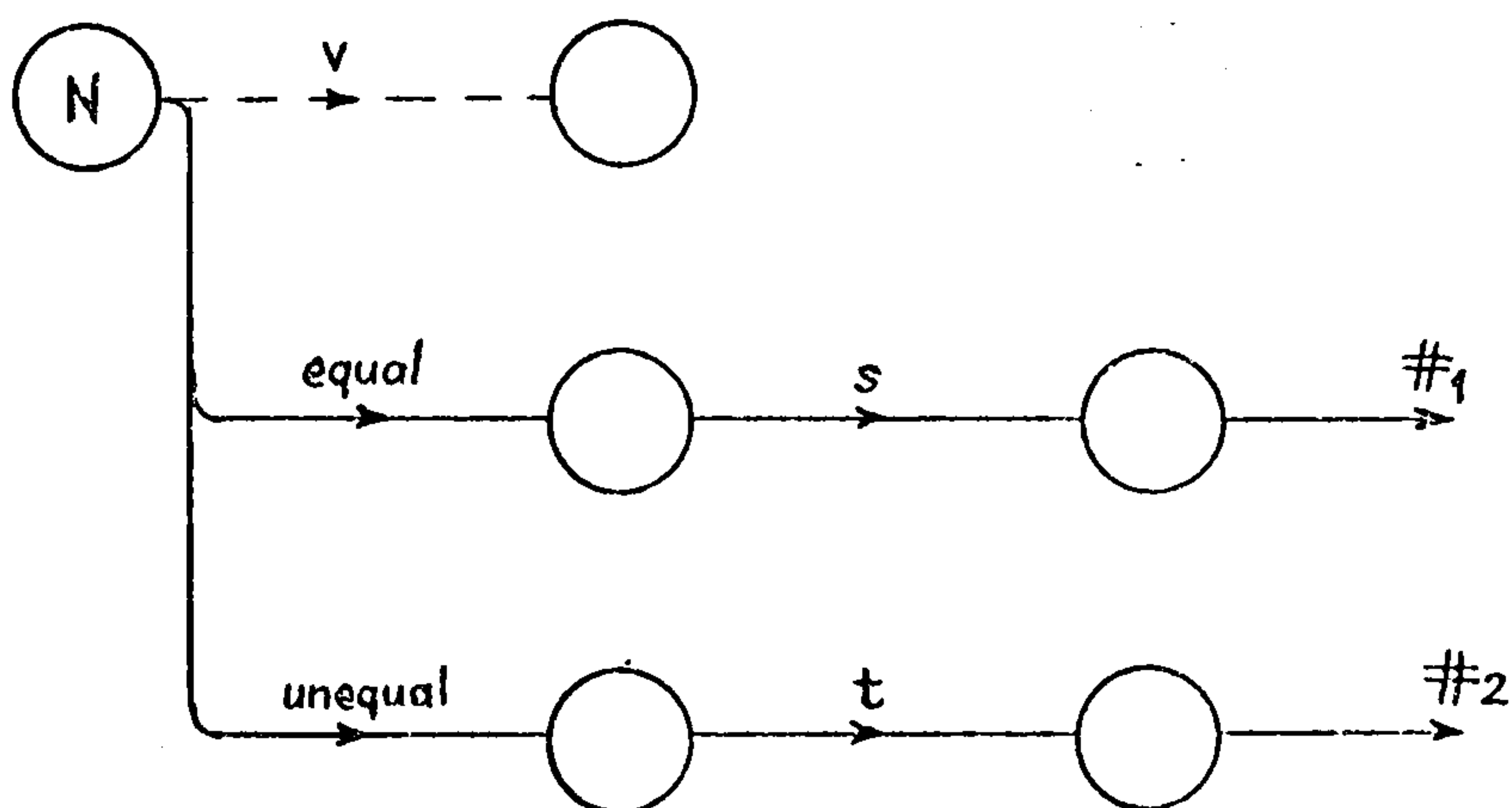


Figure 2.12 (b) Part of an AFSM constructed from the above AG fragment.

For every $a, b \in L(T)$,
 $\tilde{F}_{\text{equal}}(a, b) = \omega \vee \tilde{F}_{\text{unequal}}(a, b) = \omega .$

State N is therefore a deterministic multi-predicate state.

Control:-

v	2	$\{t, t\}$	$\{T, T\}$	
equal	2	$\{t, t\}$	$\{T, T\}$	$\lambda x \lambda y \{x=y\}$
unequal	2	$\{t, t\}$	$\{T, T\}$	$\lambda x \lambda y \{x \neq y\}$
truncate	2	$\{t, s\}$	$\{T, T\}$	$\lambda x \lambda y \{x=cy\}$

Meta-rules:-

(p1) $v(T1, T2) : \text{equal}(T1, T2) s ;$

(p2) $\text{unequal}(T1, T2) \text{truncate}(T2, T3) v(T1, T3) t .$

Figure 2.13 (a) A fragment of an AG.

(p1) $\left(\begin{smallmatrix} v \\ T1 \ T2 \end{smallmatrix} \right) : \left(\begin{smallmatrix} \text{equal} \ s \\ T1 \ T2 \end{smallmatrix} \right) ;$

(p2) $\left(\begin{smallmatrix} \text{unequal} \ \text{truncate} \ t \ v \ t \\ T1 \ T2 \quad T3 \quad T1 \ T3 \end{smallmatrix} \right) .$

(p3) $\left(\begin{smallmatrix} t \\ T1 \ T2 \ T3 \ T1 \ T3 \end{smallmatrix} \right) : \left(\begin{smallmatrix} T1 \ T2 \ T3 \end{smallmatrix} \right) .$

Figure 2.13 (b) Auxiliary grammar meta-rules corresponding to the above AG meta-rules.

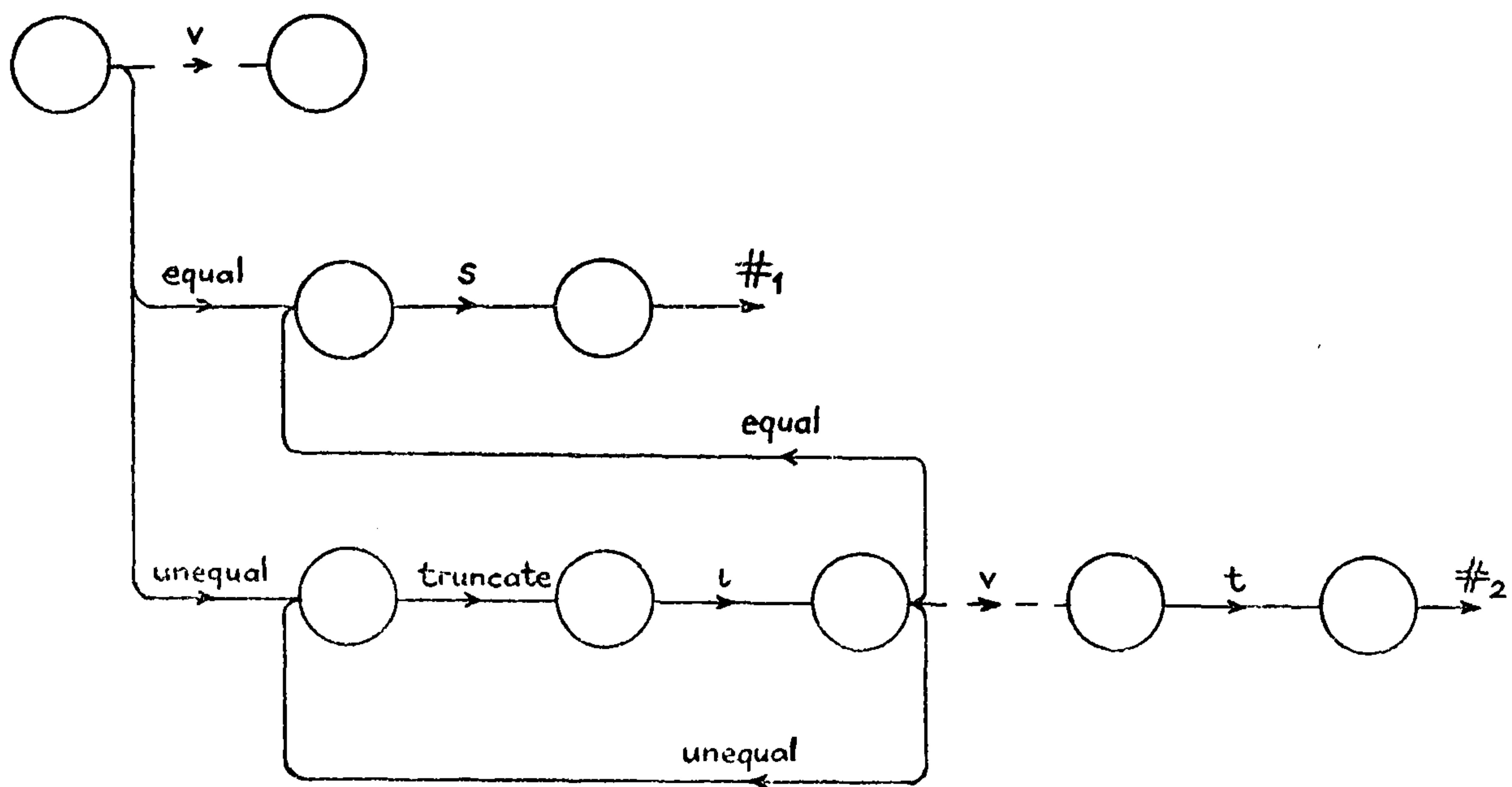


Figure 2.13 (c) Part of an AFSM constructed from the above auxiliary grammar fragment.

2.10. Summary: Construction of an AF-LR(k) Parser

Parts of this chapter, particularly sections 2.2, 2.3 and 2.5, have consisted largely of definitions, theorems and proofs, which may have tended to obscure the significance of their results. In this section, therefore, we attempt to summarise and put into perspective the main results of this chapter by outlining how one would go about constructing an AF-LR(k) parser from a well-formed affix grammar.

The following algorithm, applied in turn to each meta-rule of a well-formed AG G , directly constructs an optimised auxiliary grammar G_A of G . The sets I (containing the copy-symbols of G_A) and P_A (containing the meta-rules of G_A) are assumed to be initially empty. The algorithm uses two string variables ρ and ν , in which are built up the right-side head and tail strings (respectively) of the auxiliary grammar meta-rule.

Step 1. Let $v(\theta; \kappa)$ be the left-side hypernotation of the AG meta-rule. Set ρ to λ and ν to θ .

Step 2. If all items (hypernotations and terminals) on the right side of the AG meta-rule have already been considered, go to step 4. Otherwise, consider the leftmost item which has not yet been considered, and go to step 3.

Step 3. If the considered item is a terminal symbol, say t , replace ρ by ρt and go to step 2. Otherwise, the item must be a hypernotation, say $x(\xi; \eta)$. If ξ is a suffix of ν , replace ρ by ρx and ν by $\nu \eta$, and go to step 2. If ξ is not a suffix of ν , form the copy-meta-rule

$$\begin{pmatrix} \iota \\ \nu \xi \end{pmatrix} : \begin{pmatrix} \lambda \\ \nu \end{pmatrix} \text{ (where } \iota \text{ is a copy-symbol not already in } I \text{).}$$

If (see section 2.4) after simplification this meta-rule is "similar" to any copy-meta-rule already in P_A , replace ι by the left-side head of that copy-meta-rule;

otherwise introduce ι into I and the copy-meta-rule into P_A . Replace ρ by $\rho\iota x$ and ν by $\nu\xi\eta$, and go to step 2.

Step 4. Add the meta-rule

$$\left(\begin{smallmatrix} \nu \\ \theta_k \end{smallmatrix} \right) : \left(\begin{smallmatrix} \rho \\ \nu \end{smallmatrix} \right)$$

to P_A .

The production rules of the head grammar of G_A can now be formed from the meta-rules of G_A by removing their left- and right-side tail strings, and adding a production rule $q \rightarrow \lambda$ for each primitive predicate symbol q .

The CFSM of the head grammar may be computed by the method given by DeRemer (DeRemer 71). For each inadequate state of the CFSM (if any) the k -symbol look-ahead sets are computed (trying $k=1$, $k=2$, etc., up to some maximum acceptable value). If in any inadequate state the k -symbol look-ahead sets are not mutually disjoint, state-splitting may be tried (DeRemer 69).

The CFSM is simplified to an AFSM by eliminating each transition under a $\#$ -symbol associated with a copy-symbol or primitive predicate symbol s , and transferring any look-ahead set associated with that transition to the transition out of the same state under s .

If the AFSM has no inadequate states, the AF-LR(0) parsing algorithm of section 2.6 is applicable; or, if all its inadequate states are k -look-ahead (where $k>0$), or "better", then the more general AF-LR(k) algorithm of section 2.7 is used. Any multi-predicate states in the AFSM should be inspected to see if our informal enhancement of section 2.9 is useful.

We expect that many practical programming languages could be defined by AGs which have AF-LALR(1) auxiliary grammars, for which both our constructor and our parser are reasonably efficient.

It is relevant to compare our syntax-directed parsing technique for AGs with previous efforts along these lines.

Koster (Koster 70) uses a full top-down parsing technique, in which inherited affixes are passed down from goal (nonterminal) to sub-goal (nonterminal or primitive predicate), and derived affixes are passed up from sub-goal to goal. Left recursion cannot be handled at all by this method; and another problem is unnecessary backtracking. Under certain circumstances these problems can be eliminated by transformations to the grammar, using generalisations of methods previously developed for CFGs (Foster 68). We suspect that AGs which can be parsed deterministically from left to right in a top-down manner are those which have auxiliary grammars whose head grammars are $LL(k)$ (Knuth 71b).

Crowe's constructor (Crowe 72) generates parsers in a generalised form of Floyd Production Language. Unfortunately the generated parsers are inefficient, mainly as a result of unnecessary checking of the contents of the stack in many situations. Crowe uses a method very similar to our use of copy-symbols to bring inherited affixes to the top of the stack, and he handles left recursion (involving inherited affixes) in a manner also rather similar to ours, and with the same restrictions. But whereas our ability to handle certain left-recursive constructs is a by-product of our first optimisation of section 2.4, whose more general purpose is to eliminate unnecessary copying of affixes in the stack, Crowe's method of handling these constructs is not applicable in any other situation. Crowe relates the class of AGs accepted by his constructor to the (m,k) bounded-context grammars; we believe that in fact the class of acceptable AGs is a subset of those which have auxiliary grammars whose head grammars are (m,k) bounded-context.

Since the class of $LR(k)$ grammars properly includes both the class of $LL(k)$ grammars and the class of (m,k) bounded-context grammars, we have reason to believe that our parsing technique is more widely applicable than any other deterministic parsing method for affix grammars.

[illegible]

Comparing this new meta-rule with the corresponding meta-rule in AG form (section 1.0), we see that the nonterminal hypernotations are still there and have the same heads ('assignation', 'destination', 'source'); but the affix-positions of the first two contain 'reference to MODE', which would be illegal in an AG. Although the domain of each of these affix-positions is the set of terminal productions of 'MODE', this meta-rule can generate only production rules in which each of these affix-positions is occupied by a terminal production of 'reference to MODE' - which is precisely what we wish to specify. Since 'reference to MODE' can be derived using the upper-level rules from 'MODE', each affix-position in each production rule is guaranteed to be occupied by an affix which is within the domain of that affix-position. For example, replacing 'MODE' by 'real' throughout the meta-rule yields the production rule

```

assignation(reference to real)  : destination(reference to real)
                                becomes-symbol source(real) .

```

A further comparison of our new meta-rule with the AG meta-rule (section 1.0) shows that the hypernotation whose head was the primitive predicate symbol 'check-ref' has been dropped. Its purpose was to ensure that the affix of 'destination' was the affix of 'source' prefixed by 'reference to'. In our meta-rule this same restriction is implicitly enforced by the occurrence of 'reference to MODE' in the affix-position of 'destination'.

It turns out in fact that our extension allows primitive predicates to be dispensed with altogether, without sacrificing any of the formal power of AGs. We call our new form of two-level grammar an "extended affix grammar".

The next three sections of this chapter parallel the corresponding sections of chapter 1. In section 3.1 we give a formal definition of an extended affix grammar, and in section 3.2 a definition of well-formedness. In section 3.3 we investigate the formal properties of extended affix grammars.

In section 3.4 we show how any extended affix grammar can be converted automatically into an equivalent AG. This is of the greatest importance, as it means that our approach to parser construction from AGs, which we developed in chapter 2, is applicable to extended affix grammars as well.

We conclude in section 3.5 by discussing the results of this chapter.

3.1. Definition of an Extended Affix Grammar

We define an extended affix grammar (EAG) to be a 10-tuple

$$G = (V_n, V_t, A_n, A_t, e, R, B, D, S, P)$$

whose elements are defined in the following paragraphs.

V_t is the set of terminal symbols, V_n the set of nonterminal symbols, A_t the set of affix-terminal symbols, A_n the set of affix-nonterminal symbols, e the distinguished nonterminal, R the set of affix-rules, B the set of affix-variables, and D a map from B into A_n . These all play similar roles to the corresponding elements of an AG (see section 1.1). As in an AG, we define, for each affix-nonterminal a , $L(a)$ to be the language generated by the CFG $G_a = (A_t, A_n, a, R)$; we define L , the set of affixes, to be the union of all the sets $L(a)$ such that $a \in A_n$.

S is the control of the EAG, a set of 4-tuples

$$S_v = (v, N_v, \tau_v, \alpha_v) \quad ,$$

one for each nonterminal v . In this, N_v is the number of affix-positions of v , τ_v is a N_v -tuple over $\{1, \delta\}$ specifying the types of the affix-positions of v , and α_v is a N_v -tuple over A_n specifying the domains of the affix-positions of v . The EAG control plays a similar role to an AG control, but is simplified by the removal of the associated functions of the latter, since these are irrelevant in an EAG.

P is the set of meta-rules. Each meta-rule is of the form

$$Z : Z_1 \dots Z_m \quad (\text{where } m \geq 0)$$

Z is known as the left side, and $Z_1 \dots Z_m$ as the right side, of this meta-rule. Z is a hypernotation; and, for each j in $[1, m]$, Z_j is either a hypernotation or a terminal symbol. A hypernotation is of the form $v(f_1, \dots, f_{N_v})$, where v is a nonterminal and, for each i in $[1, N_v]$, f_i is an affix-form: f_i is a string of affix-variables and affix-terminals such that, if each affix-variable b which occurs in f_i is replaced by $D(b)$, then the resulting string of affix-nonterminals and -terminals can be derived from $\alpha_{v,i}$ using the affix-rules in R .

If the result of replacing, throughout a meta-rule

$$Z : Z_1 \dots Z_m,$$

each affix-variable b by an affix in its domain $L(D(b))$ is the rule

$$Y : Y_1 \dots Y_m,$$

then we write $Y \rightarrow Y_1 \dots Y_m$, and call this relation a production rule. We say that $Y_1 \dots Y_m$ is a direct production of Y .

A protonotion is of the form $v(g_1, \dots, g_{N_v})$, where v is an affix-nonterminal and, for each i in $[1, N_v]$, $g_i \in L(\alpha_{v,i})$. A notion is a protonotion which has at least one direct production. Thus, in a production rule $Y \rightarrow Y_1 \dots Y_m$, Y is a notion and each Y_j is either a protonotion or a terminal symbol. In particular, the distinguished nonterminal e is a notion. A blind alley is a protonotion which is not a notion.

A production of a notion X is either (i) a direct production of X , or (ii) a string of protonotions and terminals obtained by replacing, in a production of X , some notion Y by a direct production of Y . A terminal production of a notion is one which consists entirely of terminals. A sentence is a terminal production of the distinguished nonterminal e . The language of the EAG is the set of all sentences in the EAG.

$V_n = \{ \text{program, stmts, stmt, vble, tag, identify} \}$

$V_t = \{ \underline{\text{begin}}, \underline{\text{end}}, \underline{i}, :=, [,], x, y, z \}$

$A_n = \{ \text{TAG, MODE, LIST} \}$

$A_t = \{ x, y, z, i, b, a, + \}$

$e = \text{program}$

R:-

(r1-r3) TAG : x ; y ; z .

(r4-r6) MODE : i ; b ; a MODE .

(r7-r8) LIST : LIST + TAG MODE ; .

$B = \{ T, T1, M, M1, L \}$

$D = \{ (T, \text{TAG}), (T1, \text{TAG}), (M, \text{MODE}), (M1, \text{MODE}), (L, \text{LIST}) \}$

$S = \{ (\text{program}, 0, -, -),$
 $(\text{stmts}, 1, \underline{i}, \text{LIST}),$
 $(\text{stmt}, 1, \underline{i}, \text{LIST}),$
 $(\text{vble}, 2, (\underline{i}, \delta), (\text{LIST}, \text{MODE})),$
 $(\text{tag}, 1, \delta, \text{TAG}),$
 $(\text{identify}, 3, (\underline{i}, \underline{i}, \delta), (\text{LIST}, \text{TAG}, \text{MODE})) \}$

P:-

(p1) program : begin stmts(+xb+yi+zab) end .

(p2) stmts(L) : stmt(L) ;

(p3) stmts(L) i stmt(L) .

(p4) stmt(L) : vble(L, M) := vble(L, M) .

(p5) vble(L, M) : tag(T) identify(L, T, M) ;

(p6) vble(L, a M) [vble(L, i)] .

(p7) tag(x) : x .

(p8) tag(y) : y .

(p9) tag(z) : z .

(p10) identify(L + T M, T, M) : .

(p11) identify(L + T1 M1, T, M) : identify(L, T, M) .

Figure 3.1. An extended affix grammar.

This EAG defines a language of assignments between variables of identical mode. Allowable modes are defined by r4-r6: 'i' stands for 'integer', 'ab' for 'array of boolean', etc.

program

⇒ begin stmts(+xb+yi+zab) end

⇒ begin stmt(+xb+yi+zab) end

⇒ begin vble(+xb+yi+zab, b) := vble(+xb+yi+zab, b) end

⇒ begin vble(+xb+yi+zab, b) := vble(+xb+yi+zab, ab) [
vble(+xb+yi+zab, i)] end

⇒ begin vble(+xb+yi+zab, b) := vble(+xb+yi+zab, ab) [
tag(y) identify(+xb+yi+zab, y, i)] end

⇒ begin vble(+xb+yi+zab, b) := vble(+xb+yi+zab, ab) [
tag(y) identify(+xb+yi, y, i)] end

⇒ begin vble(+xb+yi+zab, b) := vble(+xb+yi+zab, ab) [
tag(y)] end

⇒ begin vble(+xb+yi+zab, b) := vble(+xb+yi+zab, ab) [y]
end

⇒ begin vble(+xb+yi+zab, b) := tag(z) identify(+xb+yi+zab,
z, ab) [y] end

⇒ begin vble(+xb+yi+zab, b) := tag(z) [y] end

⇒ begin vble(+xb+yi+zab, b) := z [y] end

⇒ begin tag(x) identify(+xb+yi+zab, x, b) := z [y] end

⇒ begin tag(x) identify(+xb+yi, x, b) := z [y] end

⇒ begin tag(x) identify(+xb, x, b) := z [y] end

⇒ begin tag(x) := z [y] end

⇒ begin x := z [y] end

Figure 3.2. A derivation in the EAG of figure 3.1.

The foregoing definition of an EAG differs from that of an AG in only two essential respects - the elimination of the primitive predicate symbols and their associated functions, and the change in the definition of a hypernotation. These two differences are however of the greatest significance.

Figure 3.1 shows an example of an EAG, and figure 3.2 an example of a derivation of a sentence in that EAG. The notational conventions we use are similar to those we used in our AG examples. (See section 1.1.)

Observe that in an EAG the distinction between the roles of the affix-variables and the affix-nonterminals is absolutely essential. A construction like the one we quoted in section 1.1 to remove affix-variables is not possible in an EAG because it would yield affix-forms which cannot be derived from the affix-nonterminals specifying the domains of their positions.

3.2. Well-formed Extended Affix Grammars

For the same reason that we defined well-formed AGs, namely to define what conditions are necessary for grammars to be well suited to parsing, we wish to define a class of "well-formed" EAGs. It turns out that, as a consequence of our definition of an EAG, only two conditions are required to ensure that an EAG is well-formed for our purposes.

A defining occurrence of an affix-variable in a meta-rule is an occurrence of that variable in an inherited affix-position of the hypernotation on the left side, or in a derived affix-position of a hypernotation on the right side, of the meta-rule. An applied occurrence of an affix-variable is an occurrence of that variable which is not a defining occurrence.

A well-formed extended affix grammar is one in which

- (a) for every affix-nonterminal a , $G_a = (A_t, A_n, a, R)$ is

unambiguous;

and (b) every variable occurring in a meta-rule has at least one defining occurrence in that meta-rule, and moreover, if a variable has an applied occurrence in a hypernotation Z on the right side of a meta-rule, then it has a defining occurrence on the left side of that meta-rule or in a hypernotation to the left of Z .

Condition (b) corresponds to condition c3 in the definition of a well-formed AG, but is less restrictive in that a variable may have more than one defining occurrence in an EAG meta-rule. Condition (a), as we shall see in section 3.4, is closely related to condition c2.

Our definition of a well-formed EAG was simpler to formulate than that of a well-formed AG. In general, however, it is not possible to determine whether an arbitrary EAG satisfies condition (a), since the ambiguity of an arbitrary CFG is undecidable (Hopcroft 69, theorem 14.7). This leads us to state the following theorem.

Theorem 3.1. It is undecidable whether an arbitrary extended affix grammar is well-formed.

As in the case of an AG, it is to be expected that any practical EAG will be written in such a way that well-formedness can in fact be determined by inspection.

The example EAG of figure 3.1 is well-formed.

3.3. Formal Properties of Extended Affix Grammars

In order to show that EAGs are as powerful as AGs, we derive in this section some results about the formal properties of EAGs to compare with the results of section 1.3.

Theorem 3.2. For every Turing machine T there exists an extended affix grammar which generates the language recognised by T .

Proof. We construct an EAG G which simulates the action of T . Each machine configuration (q, α, β) is represented by the protonotion $q(\alpha, \beta)$. A change of configuration will be represented by a direct production which simply replaces one protonotion (notation) by another.

A_t consists of the tape symbols of T . $A = \{\text{LEFT}, \text{RIGHT}, \text{SYMBOL}\}$. R contains one affix-rule $\text{SYMBOL} : s$ for each s in A_t , plus the affix-rules

LEFT : ; LEFT SYMBOL .
RIGHT : ; SYMBOL RIGHT .

V_t consists of the input symbols of T , i.e. a subset of A_t . V_n contains one symbol q for each possible state of T , with associated control

$S_q = (q, 2, (\downarrow, \downarrow), (\text{LEFT}, \text{RIGHT}))$,

plus the distinguished nonterminal e , plus one extra symbol, 'init', with associated control

$S_{\text{init}} = (\text{init}, 1, \downarrow, \text{RIGHT})$.

B contains the variable 'LEFT', 'RIGHT' and 'SYMBOL', each associated with the affix-nonterminal of the same name.

The T -rules are transliterated into meta-rules as follows.

T-rule	$\delta(q_i, s_i) = (q_f, s_f, R)$	
Meta-rule	$q_i(\text{LEFT}, s_i \text{ RIGHT}) : q_f(\text{LEFT } s_f, \text{RIGHT})$	
Effect	$\left\{ \begin{array}{c} \dots\dots\dots s_i s \dots\dots\dots \\ \uparrow \end{array} \right\} \vdash \left\{ \begin{array}{c} \dots\dots\dots s_f s \dots\dots\dots \\ \uparrow \end{array} \right\}$	
T-rule	$\delta(q_i, B) = (q_f, s_f, R)$	
Meta-rule	$q_i(\text{LEFT},) : q_f(\text{LEFT } s_f,)$	
Effect	$\left\{ \begin{array}{c} \dots\dots\dots \text{blanks} \dots\dots\dots \\ \uparrow \end{array} \right\} \vdash \left\{ \begin{array}{c} \dots\dots\dots s_f \text{blanks} \dots\dots\dots \\ \uparrow \end{array} \right\}$	

T-rule	$\delta(q_i, s_i) = (q_f, s_f, L)$
Meta-rule	$q_i(\text{LEFT SYMBOL}, s_i \text{ RIGHT}) : q_f(\text{LEFT}, \text{SYMBOL } s_f \text{ RIGHT})$
Effect	$\boxed{\begin{array}{ c c c c } \hline \dots\dots & s & s_i & \dots\dots \\ \hline \end{array}} \vdash \boxed{\begin{array}{ c c c c } \hline \dots\dots & s & s_f & \dots\dots \\ \hline \end{array}}$ <div style="display: flex; justify-content: space-around; width: 100%;"> \uparrow \uparrow </div>
T-rule	$\delta(q_i, B) = (q_f, s_f, L)$
Meta-rule	$q_i(\text{LEFT SYMBOL},) : q_f(\text{LEFT}, \text{SYMBOL } s_f)$
Effect	$\boxed{\begin{array}{ c c c c } \hline \dots\dots & s & \text{blanks } \dots & \end{array}} \vdash \boxed{\begin{array}{ c c c c } \hline \dots\dots & s & s_f & \text{blanks } \dots & \end{array}}$ <div style="display: flex; justify-content: space-around; width: 100%;"> \uparrow \uparrow </div>

If q_0 is the initial state of T , then P will also contain the meta-rules

$e : \text{init}()$,
 $\text{init}(\text{RIGHT}) : q_0(, \text{RIGHT})$,

and, for each input symbol s , a meta-rule

$\text{init}(\text{RIGHT}) : \text{init}(s \text{ RIGHT}) s$.

Finally, P will contain, for each final state q , a meta-rule

$q(\text{LEFT}, \text{RIGHT}) :$.

We first prove that the machine configurations and changes in them are correctly represented by productions and productions as stated. Let α and β be arbitrary strings of tape symbols; they fall within the domains of the variables 'LEFT' and 'RIGHT' respectively. Consider, for example, the first of the four cases above. Substituting α for 'LEFT' and β for 'RIGHT' in the meta-rule

$q_i(\text{LEFT}, s_i \text{ RIGHT}) : q_f(\text{LEFT } s_f, \text{RIGHT})$,

we obtain the production rule $q_i(\alpha, s_i\beta) \rightarrow q_f(\alpha s_f, \beta)$.

It can be seen that this production rule cannot be generated from any other meta-rule in P . Thus

$q_i(\alpha, s_i\beta) \Rightarrow q_f(\alpha s_f, \beta)$ if and only if $\delta(q_i, s_i) = (q_f, s_f, R)$.

But, by definition of T , $(q_i, \alpha, s_i\beta) \vdash (q_f, \alpha s_f, \beta)$ if and only if $\delta(q_i, s_i) = (q_f, s_f, R)$. Similar arguments apply in the other three cases. In general, therefore,

$q_i(\alpha_i, \beta_i) \Rightarrow q_f(\alpha_f, \beta_f)$ if and only if

$(q_i, \alpha_i, \beta_i) \vdash (q_f, \alpha_f, \beta_f)$. It follows by induction that

$q_1(\alpha_1, \beta_1) \Rightarrow^* q_2(\alpha_2, \beta_2)$ if and only if

$(q_1, \alpha_1, \beta_1) \vdash^* (q_2, \alpha_2, \beta_2)$. In particular,
 $q_0(\lambda, \tau) \Rightarrow^* q(\alpha, \beta)$ if and only if $(q_0, \lambda, \tau) \vdash^* (q, \alpha, \beta)$.

Now, for any string τ of input symbols,
 $e \Rightarrow \text{init}(\lambda) \Rightarrow^* \text{init}(\tau) \tau \Rightarrow q_0(\lambda, \tau) \tau$; and $q(\alpha, \beta) \Rightarrow \lambda$
 if and only if q is a final state of T . Thus,
 $e \Rightarrow^* q_0(\lambda, \tau) \tau \Rightarrow^* q(\alpha, \beta) \tau \Rightarrow \tau$ if and only if q is a
 final state of T and $(q_0, \lambda, \tau) \vdash^* (q, \alpha, \beta)$. That is, τ
 is a sentence of G if and only if T accepts τ . The theorem
 follows immediately by a generalisation of this deduction.

Corollary 3.3. Every recursively enumerable set can be generated
 by an extended affix grammar.

Note that the EAG which simulates the Turing machine is well-
 formed.

3.4. Conversion of an Extended Affix Grammar into an Equivalent Affix Grammar

We have stated that one of the constraints in our development
 of EAGs was to retain the suitability for parsing of AGs. In
 this section we show that any EAG may be automatically converted
 into an equivalent AG, and moreover that any well-formed EAG may be
 automatically converted into an equivalent well-formed AG. Thus the
 techniques of parser construction already developed for AGs can be
 applied to EAGs as well.

We use here an extension to our notation: we allow the map
 D to take as an argument any affix-form f , and define $D(f)$ to be
 the result of replacing each affix-variable which occurs in f by
 its associated affix-nonterminal. Thus, in every hypernotation
 $v(f_1, \dots, f_N)$, $\alpha_{v,i} \Rightarrow^* D(f_i)$ for each i in $[1, N]$.

Consider the EAG

$$G = (V_n, V_t, A_n, A_t, e, R, B, D, S, P) \quad .$$

We shall convert G into an AG

$$G' = (V_n, V_t, A_n, A_t, Q, e, R, B', D', S', P')$$

where $B' \supseteq B$, $D' \supseteq D$, and, if $(v, N, \tau, \alpha) \in S$, then $(v, N, \tau, \alpha, -) \in S'$.

We assume an arbitrary numbering of the affix-rules of R .

Let the r -th affix-rule be

$$a : t_0 a_1 t_1 \dots a_K t_K \quad ,$$

where $a, a_1, \dots, a_K \in A_n$ and $t_0, t_1, \dots, t_K \in A_t^*$. Then Q will contain

(i) a symbol ' synth_r ' whose associated control is

$$S'_{\text{synth}_r} = (\text{synth}_r, K+1, (\iota, \dots, \iota, \delta), (a_1, \dots, a_K, a), \lambda x_1 \dots \lambda x_K \lambda x (x = t_0 x_1 t_1 \dots x_K t_K)) \quad ;$$

and (ii) a symbol ' anal_r ' whose associated control is

$$S'_{\text{anal}_r} = (\text{anal}_r, K+1, (\iota, \delta, \dots, \delta), (a, a_1, \dots, a_K), \lambda x \lambda x_1 \dots \lambda x_K (x = t_0 x_1 t_1 \dots x_K t_K)) \quad .$$

We call the primitive predicate symbols ' synth_r ' synthesise-predicates and the symbols ' anal_r ' analyse-predicates.

P' will consist of the meta-rules of P , each transformed into AG form by the following procedure P_1 (which may as a side-effect require B' to be expanded, and D' accordingly).

Procedure P_1 . To replace the EAG meta-rule

$$\pi = \quad Z \quad : \quad Z_1 \dots Z_m$$

by an equivalent AG meta-rule.

If every affix-position of every hypernotation in π is occupied by an affix-variable whose associated affix-nonterminal specifies the domain of that affix-position, then the procedure is terminated.

Otherwise, suppose $H = v(f_1, \dots, f_N)$ occurs in π such that $D(f_i) \neq \alpha_{v,i}$. There are four possible cases:-

- (1) $H = Z_j$ (i.e. H is on the right side of π), and $\tau_{v,i} = \iota$;
- (2) $H = Z_j$, and $\tau_{v,i} = \delta$;
- (3) $H = Z$ (i.e. H is on the left side of π), and $\tau_{v,i} = \iota$;
- (4) $H = Z$, and $\tau_{v,i} = \delta$.

By definition of a hypernotation, $\alpha_{v,i} \Rightarrow^* D(f_i)$. Since $\alpha_{v,i} \neq D(f_i)$, let the last affix-rule applied in a derivation $\alpha_{v,i} \Rightarrow^* D(f_i)$ be

$$a : t_0 a_1 t_1 \dots t_K a_K \quad ,$$

where $a, a_1, \dots, a_K \in A_n$ and $t_0, t_1, \dots, t_K \in A_t^*$, and let this be the r -th affix-rule of R . Then there must exist $\theta, \eta \in (A_n \cup A_t)^*$ such that

$$\alpha_{v,i} \Rightarrow^* \theta a \eta \Rightarrow \theta t_0 a_1 t_1 \dots a_K t_K \eta = D(f_i),$$

and therefore f_i must be of the form

$$f_i = \mu t_0 b_1 t_1 \dots b_K t_K \nu \quad ,$$

where $D(\mu) = \theta$, $D(\nu) = \eta$, and b_k is an affix-variable with $D(b_k) = a_k$ for each k in $[1, K]$. Let b be a variable, with $D(b) = a$, not occurring in π . Then a new meta-rule π' is formed from π using one of the following four transformations, according to which of the above four cases holds:-

$$(T1) \quad \pi' = \quad Z : \quad Z_1 \dots Z_{j-1} \text{ synth}_r(b_1, \dots, b_K, b) \\ v(f_1, \dots, f_{i-1}, \mu b \nu, f_{i+1}, \dots, f_N) \\ Z_{j+1} \dots Z_m \quad .$$

$$(T2) \quad \pi' = \quad Z : \quad Z_1 \dots Z_{j-1} \\ v(f_1, \dots, f_{i-1}, \mu b \nu, f_{i+1}, \dots, f_N) \\ \text{anal}_r(b, b_1, \dots, b_K) Z_{j+1} \dots Z_m \quad .$$

$$(T3) \quad \pi' = \quad v(f_1, \dots, f_{i-1}, \mu b \nu, f_{i+1}, \dots, f_N) : \\ \text{anal}_r(b, b_1, \dots, b_K) Z_1 \dots Z_m \quad .$$

$$(T4) \quad \pi' = \quad v(f_1, \dots, f_{i-1}, \mu b \nu, f_{i+1}, \dots, f_N) : \\ Z_1 \dots Z_m \text{ synth}_r(b_1, \dots, b_K, b) \quad .$$

π is replaced by π' , and the procedure is repeated.

Theorem 3.4. The above method converts an extended affix grammar G into an affix grammar G' which is loosely equivalent to G .

Proof. P1 always terminates since, for each affix form f , the

derivation of $D(f)$ contains a finite number of steps, and each of the transformations T1-T4 shortens such a derivation by one step. The terminating condition of P1 ensures that every affix-position of every hypernotation in a meta-rule is occupied by an appropriate affix-variable, so the resulting meta-rule is in fact an AG meta-rule.

It remains to be proved that the meta-rule π' resulting from any of the transformations T1-T4 is in some way equivalent to the meta-rule π . Note that it is possible that an intermediate meta-rule is neither an EAG meta-rule (because of the presence of primitive predicate hypernotations) nor an AG meta-rule (because of the presence of generalised affix-forms), although of course the original is an EAG meta-rule and the final one is an AG meta-rule. To handle this possibility, we informally allow "generalised" meta-rules, which are like EAG meta-rules but may contain primitive predicate hypernotations, with production rules for primitive predicate protonotions being defined in the same way as in AGs. Thus EAG meta-rules and AG meta-rules are particular cases of "generalised" meta-rules.

Consider, for example, the "generalised" meta-rules π and π' before and after transformation T4. In this case,

$$\begin{aligned}\pi &= v(f_1, \dots, f_{i-1}, \mu_{t_0}^{b_1} t_1 \dots b_K^{t_K} v, f_{i+1}, \dots, f_N) : \\ &\quad Z_1 \dots Z_m, \\ \pi' &= v(f_1, \dots, f_{i-1}, \mu b v, f_{i+1}, \dots, f_N) : \\ &\quad Z_1 \dots Z_m \text{ synth}_r(b_1, \dots, b_K, b) .\end{aligned}$$

Let g_1, \dots, g_K be arbitrary affixes in the respective domains of the variables b_1, \dots, b_K . Substituting these affixes for these variables in π , and arbitrary affixes for any other variables occurring in π , we obtain a production rule of the form

$$\rho = v(h_1, \dots, h_{i-1}, \phi_{t_0}^{g_1} t_1 \dots g_K^{t_K} v, h_{i+1}, \dots, h_N) \rightarrow Y_1 \dots Y_m .$$

Substituting the same affixes for the same variables in π' , we obtain

$$\frac{v(h_1, \dots, h_{i-1}, \phi b \gamma, h_{i+1}, \dots, h_N)}{\text{synth}_r(g_1, \dots, g_K, b)} : Y_1 \dots Y_m,$$

since by construction b does not occur in any of $f_1, \dots, f_{i-1}, f_{i+1}, \dots, f_N, \mu, \nu, Z_1, \dots, Z_m$. Further substituting for b an arbitrary affix g in the domain of b , we obtain the production rule

$$\rho' = \frac{v(h_1, \dots, h_{i-1}, \phi g \gamma, h_{i+1}, \dots, h_N)}{Y_1 \dots Y_m \text{ synth}_r(g_1, \dots, g_K, g)}.$$

Now, if $g \neq t_0 g_1 t_1 \dots g_K t_K$, then $\text{synth}_r(g_1, \dots, g_K, g)$ is a blind alley, and therefore ρ' cannot be applied in a derivation of any sentence. On the other hand, if $g = t_0 g_1 t_1 \dots g_K t_K$, then $\text{synth}_r(g_1, \dots, g_K, g) \rightarrow \lambda$, and therefore

$$\begin{aligned} & v(h_1, \dots, h_{i-1}, \phi t_0 g_1 t_1 \dots g_K t_K \gamma, h_{i+1}, \dots, h_N) \\ & \Rightarrow Y_1 \dots Y_m \text{ synth}_r(g_1, \dots, g_K, t_0 g_1 t_1 \dots g_K t_K) \\ & \Rightarrow Y_1 \dots Y_m. \end{aligned}$$

Thus there is a one-to-one correspondence between the production rules which can usefully be generated from π and π' , and corresponding production rules cause the same notion to be replaced by the same string of protonotions and terminals, after replacement of primitive predicate notions.

Similar arguments can be used to obtain the same result for the meta-rules before and after transformations T1-T3. The theorem follows by induction.

We have established that any EAG can be converted into an equivalent AG. What is still more relevant, however, is whether a well-formed EAG can be converted into an equivalent well-formed AG.

This, in fact, can be done, and all that is necessary, beyond transformation of the EAG meta-rules by procedure P1, is to eliminate repeated defining occurrences of the same affix-variable in a given meta-rule.

For each affix-nonterminal a , we introduce into Q a new symbol ' equal_a ', whose associated control is

$$S'\text{equal}_a = (\text{equal}_a, 2, (\iota, \iota), (a, a), \lambda x \lambda y (x=y)) \quad .$$

We call the primitive predicate symbols ' equal_a ' equal-predicates.

We further transform each meta-rule of P' by applying the following procedure P2 (which may as a side-effect require B' to be further expanded, and D' accordingly).

Procedure P2. To eliminate multiple defining occurrences of affix-variables in the affix grammar meta-rule

$$\pi = \quad Z \quad : \quad Z_1 \dots\dots Z_m \quad .$$

If no affix-variable has more than one defining occurrence in π , then the procedure is terminated.

Otherwise, suppose the variable b has more than one defining occurrence. Let b' be a variable, with $D(b') = D(b) = a$, not occurring in π . There are two possible cases (which are not mutually exclusive).

(T5) b has two defining occurrences on the left side of π , i.e. Z is of the form $v(\dots, b, \dots, b, \dots)$. Form a new meta-rule π' from π as follows:-

$$\pi' = \quad v(\dots, b, \dots, b', \dots) : \text{equal}_a(b', b) Z_1 \dots\dots Z_m \quad .$$

(T6) b has a defining occurrence in Z and another in Z_j , or has a defining occurrence in Z_k and another in Z_j (where $k \leq j$). If Z_j is of the form $v(\dots, b, \dots)$ then form a new meta-rule π' from π as follows:-

$$\pi' = \quad Z \quad : \quad Z_1 \dots Z_{j-1} \quad v(\dots, b', \dots) \text{equal}_a(b', b) \quad Z_{j+1} \dots Z_m \quad .$$

Replace π by π' , and repeat the procedure.

Theorem 3.5. Application of procedures P1 and P2 to each meta-rule converts a well-formed extended affix grammar G into a well-formed affix grammar G' which is loosely equivalent to G .

Proof. The proof of theorem 3.4 can be extended to cover procedure P2 with transformations T5-T6. Thus G' is an AG and is loosely equivalent to G . It only remains to be proved that if G is a well-formed EAG then G' is a well-formed AG.

Condition c1 in the definition of a well-formed AG is obviously satisfied since, by construction, every affix-position of every hypernotation occurring in a meta-rule of G' is occupied by a variable whose associated affix-nonterminal specifies the domain of that affix-position, i.e. in every hypernotation $x(b_1, \dots, b_N)$, and for each i in $[1, N]$, b_i is a variable with $D(b_i) = \alpha_{x,i}$, regardless of whether $\tau_{x,i} = \iota$ or $\tau_{x,i} = \delta$, or whether the hypernotation is on the left or right side of a meta-rule.

The set of primitive predicate symbols of G' is

$$Q = \{ \text{anal}_r, \text{synth}_r \mid r \text{ is the number of an affix-rule} \} \cup \{ \text{equal}_a \mid a \in A_n \}.$$

The symbols ' equal_a ' have no derived affixes, so the functions $\tilde{F}_{\text{equal}_a}$ obviously exist. If affix-rule r is

$$a : t_0 a_1 t_1 \dots a_K t_K,$$

then the function mapping the inherited affixes of ' synth_r ' to its derived affix is

$$\tilde{F}_{\text{synth}_r} = \lambda x_1 \dots \lambda x_K (t_0 x_1 t_1 \dots x_K t_K)$$

and the function mapping the inherited affix of ' anal_r ' to its derived affixes is

$$\tilde{F}_{\text{anal}_r} = \lambda x (\exists x_1 \in L(a_1), \dots, x_K \in L(a_K) : x = t_0 x_1 t_1 \dots x_K t_K \mid (x_1, \dots, x_K) \mid \omega).$$

Clearly $\tilde{F}_{\text{synth}_r}(y_1, \dots, y_K) = y$ if and only if

$F_{\text{synth}_r}(y_1, \dots, y_K, y) = \underline{\text{true}}$, and $\tilde{F}_{\text{anal}_r}(y) = (y_1, \dots, y_K)$ if and only if $F_{\text{anal}_r}(y, y_1, \dots, y_K) = \underline{\text{true}}$. Moreover, $\tilde{F}_{\text{synth}_r}$

is obviously single-valued. Since G_a is unambiguous, by condition (a) in the definition of a well-formed EAG, \tilde{F}_{anal_r} is also single-valued, which we prove by contradiction. If \tilde{F}_{anal_r} were not single-valued, suppose that $\tilde{F}_{anal_r}(y) = (y_1, \dots, y_K)$ and $\tilde{F}_{anal_r}(y) = (y_1', \dots, y_K')$. For each i in $[1, K]$, $y_i \in L(a_i)$, i.e. $a_i \Rightarrow^* y_i$; therefore

$$\begin{aligned} a &\Rightarrow t_0 a_1 t_1 \dots a_K t_K \\ &\Rightarrow^* t_0 a_1 t_1 \dots y_K t_K \\ &\vdots \\ &\Rightarrow^* t_0 y_1 t_1 \dots y_K t_K \\ &= y \end{aligned}$$

Likewise,

$$\begin{aligned} a &\Rightarrow t_0 a_1 t_1 \dots a_K t_K \\ &\Rightarrow^* t_0 a_1 t_1 \dots y_K' t_K \\ &\vdots \\ &\Rightarrow^* t_0 y_1' t_1 \dots y_K' t_K \\ &= y \end{aligned}$$

If $(y_1, \dots, y_K) \neq (y_1', \dots, y_K')$, then there are two different canonical derivations $a \Rightarrow^* y$, which contradicts the unambiguity of G_a . Thus condition c2 is satisfied.

If a "generalised" meta-rule π satisfies condition (b) in the definition of a well-formed EAG, then so does the meta-rule π' resulting from any of transformations T1--T6. Consider, for example, transformation T4, involving a derived affix-position on the left side of π . Recall that

$$\pi = \begin{array}{c} v(f_1, \dots, f_{i-1}, \mu t_0 b_1 t_1 \dots b_K t_K v, f_{i+1}, \dots, f_N) \\ Z_1 \dots Z_m \end{array} :$$

and that

$$\pi' = \begin{array}{c} v(f_1, \dots, f_{i-1}, \mu b v, f_{i+1}, \dots, f_N) \\ \text{synth}_r(b_1, \dots, b_K, b) \end{array} : Z_1 \dots Z_m$$

In π , each of the variables b_1, \dots, b_K must, by condition (b), have a defining occurrence in an inherited affix-position on the left side or in $Z_1 \dots Z_m$; therefore the applied occurrence of each of these variables in the

hypernotation $\text{synth}_r(b_1, \dots, b_K, b)$ in π' satisfies condition (b). Also, the new variable b , which has an applied occurrence on the left side of π' , also has a defining occurrence, in $\text{synth}_r(b_1, \dots, b_K, b)$. Similar arguments can be applied to the other transformations.

It follows by induction that, if each meta-rule of G satisfies condition (b), then the corresponding meta-rule of G' satisfies this condition too. Thus each variable occurring in a meta-rule of G' has at least one defining occurrence in that meta-rule. But the terminating condition of procedure P2 ensures that each variable has at most one defining occurrence in any meta-rule of G' . Thus condition c3 is satisfied.

We have shown how any (well-formed) EAG can be converted into an equivalent (well-formed) AG. Thus the parsing technique we developed in chapter 2 (or any other parsing technique for AGs) can be applied to EAGs. We investigate this possibility more deeply in chapter 4.

Figure 3.3 shows the well-formed AG obtained by converting the EAG of figure 3.1 according to the transformations given in this section.

3.5. Some Comments on Extended Affix Grammars

We believe our examples (and a further example appears in the appendix) have demonstrated that EAGs are significantly more readable than AGs. Even the grammars which simulate a Turing machine (sections 1.3 and 3.3), although not practical examples, reinforce our view, and it is noteworthy that the proof of correctness of the simulation by an EAG is rather more elegant than that of the simulation by an AG.

$$\begin{aligned}
Q &= \{ \text{synth}_1, \text{synth}_2, \text{synth}_3, \text{synth}_4, \text{synth}_5, \text{synth}_6, \\
&\quad \text{synth}_7, \text{synth}_8, \text{anal}_4, \text{anal}_6, \text{anal}_7, \text{equal}_{\text{MODE}}, \\
&\quad \text{equal}_{\text{TAG}} \} \\
B' &= \{ T, T1, T2, M, M1, M2, M3, L, L1, L2, L3 \} \\
D' &= \{ (T, \text{TAG}), (T1, \text{TAG}), (T2, \text{TAG}), (M, \text{MODE}), (M1, \text{MODE}), \\
&\quad (M2, \text{MODE}), (M3, \text{MODE}), (L, \text{LIST}), (L1, \text{LIST}), (L2, \text{LIST}), \\
&\quad (L3, \text{LIST}) \} \\
S' &= \{ (\text{program}, 0, -, -, -), \\
&\quad (\text{stmts}, 1, \iota, \text{LIST}, -), \\
&\quad (\text{stmt}, 1, \iota, \text{LIST}, -), \\
&\quad (\text{vble}, 2, (\iota, \delta), (\text{LIST}, \text{MODE}), -), \\
&\quad (\text{tag}, 1, \delta, \text{TAG}, -), \\
&\quad (\text{identify}, 3, (\iota, \iota, \delta), (\text{LIST}, \text{TAG}, \text{MODE}), -), \\
&\quad (\text{synth}_1, 1, \delta, \text{TAG}, \lambda t \{ t=x \}), \\
&\quad (\text{synth}_2, 1, \delta, \text{TAG}, \lambda t \{ t=y \}), \\
&\quad (\text{synth}_3, 1, \delta, \text{TAG}, \lambda t \{ t=z \}), \\
&\quad (\text{synth}_4, 1, \delta, \text{MODE}, \lambda m \{ m=i \}), \\
&\quad (\text{synth}_5, 1, \delta, \text{MODE}, \lambda m \{ m=b \}), \\
&\quad (\text{synth}_6, 2, (\iota, \delta), (\text{MODE}, \text{MODE}), \lambda m \lambda n \{ n=am \}), \\
&\quad (\text{synth}_7, 4, (\iota, \iota, \iota, \delta), (\text{LIST}, \text{TAG}, \text{MODE}, \text{LIST}), \\
&\quad \quad \quad \lambda l \lambda t \lambda m \lambda k \{ k=l+tm \}), \\
&\quad (\text{synth}_8, 1, \delta, \text{LIST}, \lambda l \{ l=\lambda \}), \\
&\quad (\text{anal}_4, 1, \iota, \text{MODE}, \lambda m \{ m=i \}), \\
&\quad (\text{anal}_6, 2, (\iota, \delta), (\text{MODE}, \text{MODE}), \lambda m \lambda n \{ m=an \}), \\
&\quad (\text{anal}_7, 4, (\iota, \delta, \delta, \delta), (\text{LIST}, \text{LIST}, \text{TAG}, \text{MODE}), \\
&\quad \quad \quad \lambda k \lambda l \lambda t \lambda m \{ k=l+tm \}), \\
&\quad (\text{equal}_{\text{MODE}}, 2, (\iota, \iota), (\text{MODE}, \text{MODE}), \lambda m \lambda n \{ m=n \}), \\
&\quad (\text{equal}_{\text{TAG}}, 2, (\iota, \iota), (\text{TAG}, \text{TAG}), \lambda t \lambda u \{ t=u \}) \}
\end{aligned}$$

(continued)

Figure 3.3. The AG constructed from the EAG of figure 3.1 by the method described in section 3.4.

V_n, V_t, A_n, A_t, e, R are as in figure 3.1.

P' :-

(p1)	<pre> program : begin synth₂(L) synth₁(T) synth₅(M) synth₇(L,T,M,L1) synth₂(T1) synth₄(M1) synth₇(L1,T1,M1,L2) synth₃(T2) synth₅(M2) synth₆(M2,M3) synth₇(L2,T2,M3,L3) stmts(L3) end . </pre>	T1
(p2)	stmts(L) : stmt(L) ;	
(p3)	stmts(L) ; stmt(L) .	
(p4)	<pre> stmt(L) : vble(L,M) := vble(L,M1) equalMODE(M1,M) . </pre>	T6
(p5)	vble(L,M) : tag(T) identify(L,T,M) ;	
(p6)	<pre> vble(L,M1) anal₆(M1,M) [vble(L,M2) anal₄(M2)] . </pre>	T2
(p7)	tag(T) : x synth ₁ (T) ;	T4
(p8)	y synth ₂ (T) ;	T4
(p9)	z synth ₃ (T) .	T4
(p10)	<pre> identify(L1,T,M) : anal₇(L1,L,T1,M) equalTAG(T1,T) ; </pre>	T3, T6
(p11)	anal ₇ (L1,L,T1,M1) identify(L,T,M) .	T3

Figure 3.3 (concluded)

Transformations applied to each meta-rule are shown on the right.

We have certainly found by experience that writing an EAG is much less tedious and more creative than writing an AG.

The reason for these observations is undoubtedly that in an EAG relationships among affixes can be expressed in a direct and natural manner in the meta-rules themselves, and not hidden away in some primitive predicate functions. (An excellent example of this is meta-rule pl0 in figures 3.3 and 3.1.) The simple predicates which tend to obscure AG meta-rules are just not necessary in EAGs. Certainly, in a given situation, these predicates specify checks or actions which must be performed during a parse; but section 3.4 has shown that such predicates can be inserted automatically.

In one sense, the relationship between EAGs and AGs is similar to that between high-level and low-level languages. In another sense, EAGs are declarative systems whilst AGs are imperative systems.

High-level languages tend to take away some of the programming flexibility of low-level languages. EAGs have the same drawback when compared with AGs. For example, whilst one can specify that two affixes must be equal simply by using the same affix-variable in a meta-rule, there is no elegant way in an EAG of specifying that two affixes must be unequal; in an AG the necessary primitive predicate function would be just as trivial as one for ensuring equality. We shall return to this point later.

A major example of the use of an EAG to define completely the syntax of a practical programming language is given in appendix A. Some of our small examples have been extracted from this grammar, perhaps with slight modifications, to illustrate particular points.

CHAPTER 4LEFT-TO-RIGHT PARSERS FOR WELL-FORMED EXTENDED AFFIX GRAMMARS
*****4.0. Introduction

We have already assembled all the techniques necessary to construct a left-to-right parser from a suitable well-formed EAG. In section 3.4 we showed how to convert any well-formed EAG into a well-formed AG. In chapter 2 we showed how to convert a well-formed AG into an auxiliary grammar, and we defined the class of AF-LR(k) auxiliary grammars to which our LR(k)-based parsing method can be applied.

There are, however, two further, closely related, problems which must be solved before the construction of a parser from an EAG can be completely automated. We must design a parse-time representation of affixes, and we must show how to implement the synthesise-, analyse- and equal-predicate functions generated by the EAG-to-AG convertor. These problems we consider in sections 4.1 and 4.2.

In section 4.3 we consider the problem of left recursion as it applies to EAGs. This is somewhat analogous to a problem already investigated for AGs. An opportunity is taken to introduce an improvement into our EAG-to-AG convertor.

In section 4.4 we show that certain multi-predicate states in an EAG parser can be proved to be deterministic. This extends beyond AF-LR(k) the class of grammars from which we can automatically construct deterministic parsers.

In order to estimate the practical value of our results, we describe in sections 4.5 and 4.6 a machine-code implementation of our parser, including some optimisations which can be deduced from the EAG; and in section 4.7 we describe a simple empirical investigation into the efficiency of this implementation. Finally, in section 4.8, we discuss the possibility of incorporating our parser into a practical translator.

4.1. Parse-time Representation of Affixes

A suitable representation of affixes at parse time will be one which permits an efficient implementation of the primitive predicate functions of the AG. In the case of a parser constructed from an EAG, the functions to be implemented are those of the synthesise-, analyse- and equal-predicates defined in section 3.4. Recall that, if affix-rule r is

$$a : t_0 a_1 t_1 \dots a_K t_K \quad ,$$

then

$$\begin{aligned} \tilde{F}_{\text{synth}_r} &= \lambda x_1 \dots \lambda x_K (t_0 x_1 t_1 \dots x_K t_K) \quad , \\ \tilde{F}_{\text{anal}_r} &= \lambda x \left(\exists x_1 \in L(a_1), \dots, x_K \in L(a_K) : x = t_0 x_1 t_1 \dots x_K t_K \right. \\ &\quad \left. \mid (x_1, \dots, x_K) \mid \omega \right) \quad , \end{aligned}$$

and that, for each affix-nonterminal a ,

$$\tilde{F}_{\text{equal}_a} = \lambda x_1 \lambda x_2 (x_1 = x_2 \mid \phi \mid \omega) \quad ,$$

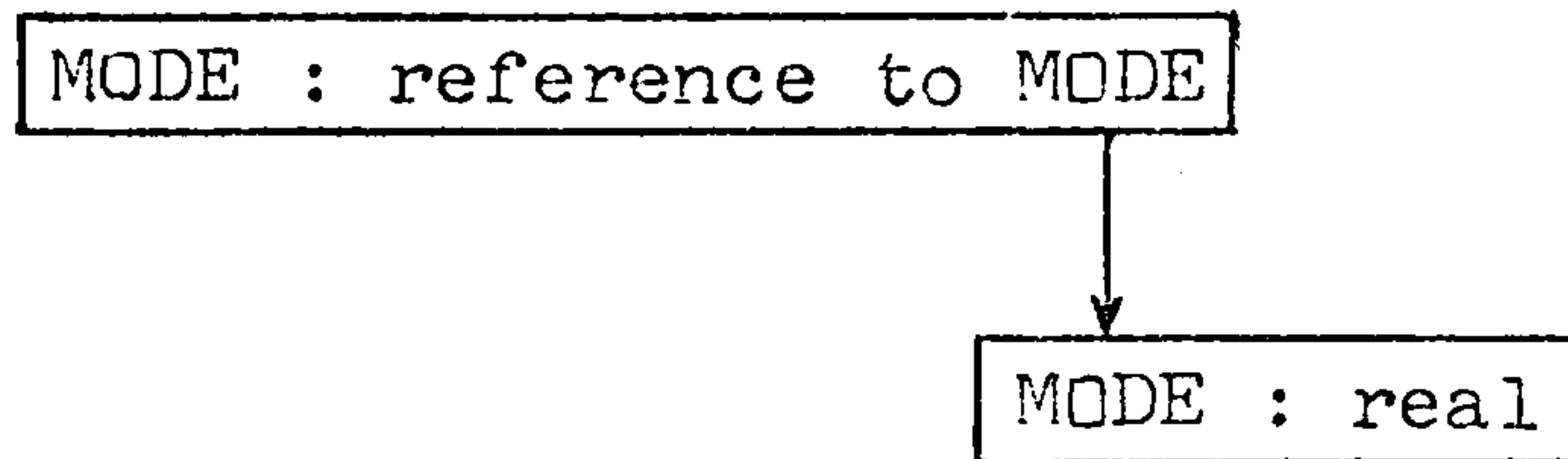
where ϕ denotes a 0-tuple.

The most obvious representation of affixes, by strings, is unsuitable in general, because each analyse-predicate function would have to perform a complete (context-free) parse of its inherited affix. (It may be mentioned in passing, however, that such a representation is worth consideration in certain special cases. For example, if every affix-rule contains at most one affix-nonterminal, each analyse-predicate function could simply check for and remove particular affix-terminals at the ends of the string representing the affix.)

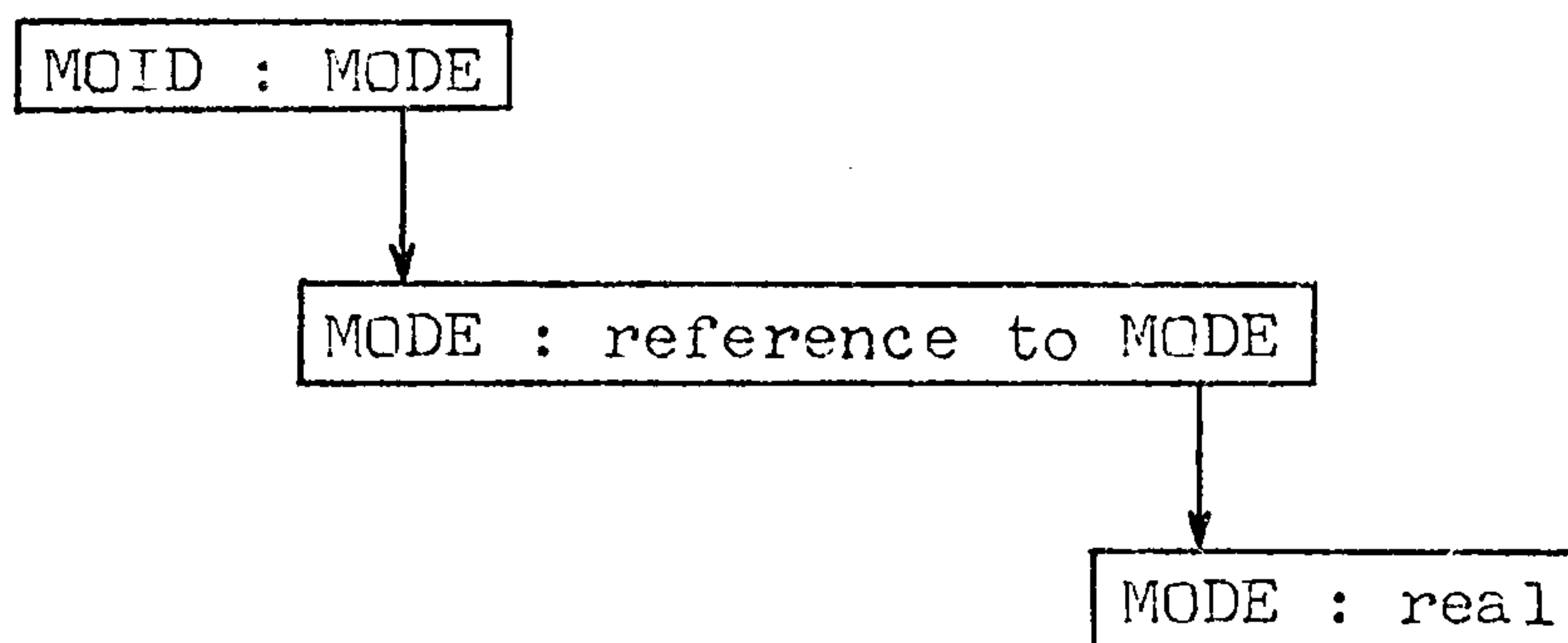
In order to avoid re-parsing of affixes, the most suitable representation of an affix which is a terminal production of an affix-nonterminal a is the derivation tree for the affix in the context-free grammar G_a . Since G_a is unambiguous, by definition of a well-formed EAG, a derivation tree for an affix as a terminal production of a particular affix-nonterminal is unique. We emphasise this point because it influenced a feature of our EAG-to-AG convertor. An affix which is simultaneously a terminal production of n affix-nonterminals may have one of n different derivation trees. This point is illustrated in figure 4.1. For this reason, our parser will work with this representation only because every affix-position of every hypernotation in the meta-rules of the AG produced by our convertor is occupied by a variable whose associated affix-nonterminal specifies the domain of that affix-position. In a well-formed AG a wider choice of variables is allowed in a given affix-position (see condition c1 in the definition of a well-formed AG - section 1.2); but, if our EAG-to-AG convertor took advantage of this fact, a variable might be assigned an affix which, although in itself a possible value of that variable, is represented by a tree with the wrong affix-nonterminal at its root; the parser would therefore malfunction. An example of this will be given in section 4.2.

Conventionally, the nodes of a derivation tree in a CFG are labelled by nonterminals and terminals. To represent an affix, we prefer to use a tree in which each node is labelled by the affix-rule applied at that node and has one direct descendent for each affix-nonterminal in the right side of that affix-rule. A derivation tree in our form could be converted into one in conventional form, and vice versa. Our form has the advantage of being more compact and is more suited to our purpose. Examples of derivation trees representing affixes appear in figures 4.1 and 4.5.

At parse time the nodes of the derivation trees will be stored separately from the affix stack, in a dynamic storage area



- (a) Derivation tree of 'reference to real' as a terminal production of 'MODE'.



- (b) Derivation tree of 'reference to real' as a terminal production of 'MOID'.

Figure 4.1 Different derivation trees for the same affix.

The affix-rules are:-

MOID : MODE ; void .

MODE : real ; reference to MODE .

or heap. Each node will be a record containing the following data: (1) the number, r , of the affix-rule labelling this node (using the same numbering of affix-rules as in the construction of section 3.4); (2) the number of affix-nonterminals, K , in the right side of affix-rule r ; and (3) pointers to the K nodes which are the direct descendants of the current node. Each affix will be represented by a pointer to the root node of its derivation tree. Thus the affix stack will be a stack of pointers.

Copying of affixes in the stack (when a copy-transition is traversed) will, of course, be implemented by copying pointers, not by reproducing the trees to which they point. This, and other aspects of the implementations to be described in section 4.2, imply that the derivation trees of affixes which exist at any stage of a parse are not necessarily disjoint, but may have common subtrees. In fact, each tree is in general just a part of a larger and more general data structure, a directed graph which has the property of containing no loops.

4.2. Implementation of Synthesise-, Analyse- and Equal-Predicate Functions

Having defined our parse-time representation of affixes, we proceed to describe the implementation of the various primitive predicate functions introduced by the EAG-to-AG convertor. As usual, we assume that the r -th affix-rule of the EAG is

$$a : t_0 a_1 t_1 \dots a_K t_K \quad .$$

Recall that

$$\tilde{F}_{\text{synth}_r} = \lambda x_1 \dots \lambda x_K (t_0 x_1 t_1 \dots x_K t_K) \quad .$$

The action of the function $y = \tilde{F}_{\text{synth}_r}(y_1, \dots, y_K)$ is illustrated in figure 4.2. A new node is created in the heap; it is labelled by affix-rule r , and has as its direct descendants the nodes to which the pointers representing y_1, \dots, y_K point; y is then

represented by a pointer to the new node. A proof of the correctness of this implementation is quite simple. For each i in $[1, K]$, the domain of the i -th parameter of $\tilde{F}_{\text{synth}_r}$ is $L(a_i)$, so $a_i \Rightarrow^* y_i$. It follows that

$$\begin{aligned} a &\Rightarrow t_0 a_1 t_1 \dots a_K t_K \\ &\Rightarrow^* t_0 a_1 t_1 \dots y_K t_K \\ &\vdots \\ &\Rightarrow^* t_0 y_1 t_1 \dots y_K t_K \\ &= y \end{aligned}$$

is the (right) derivation of y from a , and in the corresponding tree the root will be labelled by ' $a:t_0 a_1 t_1 \dots a_K t_K$ ' and the root's direct descendants will be the roots of the derivation trees corresponding to the derivations $a_1 \Rightarrow^* y_1, \dots, a_K \Rightarrow^* y_K$.

The typical analyse-predicate function is

$$\tilde{F}_{\text{anal}_r} = \lambda x \left(\begin{array}{l} \exists x_1 \in L(a_1), \dots, x_K \in L(a_K) : x = t_0 x_1 t_1 \dots x_K t_K \\ \mid (x_1, \dots, x_K) \mid \omega \end{array} \right).$$

The action of the function $\tilde{F}_{\text{anal}_r}(y) = (y_1, \dots, y_K)$ is shown in figure 4.3. If the node to which the pointer representing y points is labelled by affix-rule r , then y_1, \dots, y_K will be represented by pointers to the K direct descendants of that node; otherwise the value of the function is $\tilde{F}_{\text{anal}_r}(y) = \omega$. To prove this, note that $\tilde{F}_{\text{anal}_r}(y) = (y_1, \dots, y_K)$ if and only if

$$\begin{aligned} a &\Rightarrow t_0 a_1 t_1 \dots a_K t_K \\ &\Rightarrow^* t_0 a_1 t_1 \dots y_K t_K \\ &\vdots \\ &\Rightarrow^* t_0 y_1 t_1 \dots y_K t_K \\ &= y, \end{aligned}$$

which in turn is equivalent to saying that the node to which the pointer representing y points is labelled by ' $a:t_0 a_1 t_1 \dots a_K t_K$ ' and that a pointer to the i -th direct descendant of that node is a representation of y_i . $\tilde{F}_{\text{anal}_r}(y) = \omega$ if and only if the derivation $a \Rightarrow^* y$ does not start with $a \Rightarrow t_0 a_1 t_1 \dots a_K t_K$, i.e. if the node to

which the pointer representing y points is not labelled by that affix-rule.

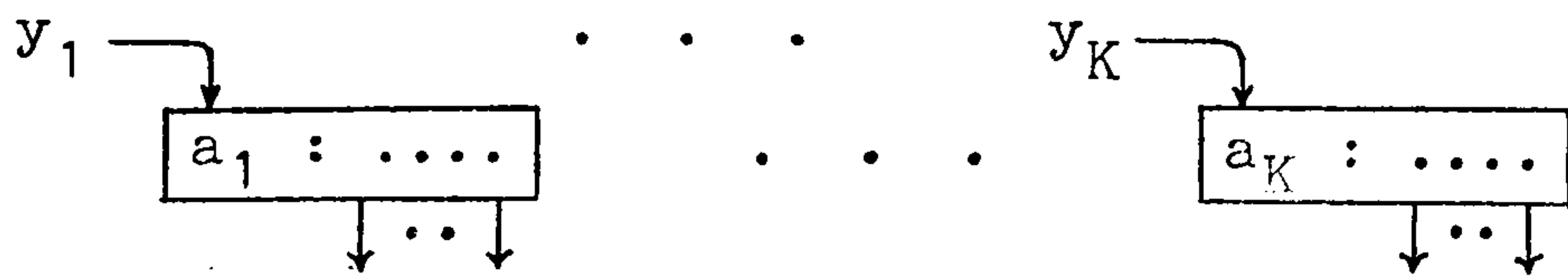
Our choice of representation for affixes has permitted a very simple and efficient implementation of the synthesise- and analyse-predicate functions. This, however, has been achieved at the expense of a more complex implementation of the equal-predicate functions. This was a deliberate design decision, as we expect the equal-predicate functions to be applied rather less frequently than the other functions.

The equal-predicate functions must be implemented by a tree-matching procedure, which will use a stack. The affix stack can be used for this purpose, provided that it is restored to its original state before exit from the procedure. The procedure should take into account the possibility that the two derivation trees may have subtrees in common. Our procedure checks the "equivalence" of the nodes to which the pointers representing the two affixes point. Two nodes are "equivalent"

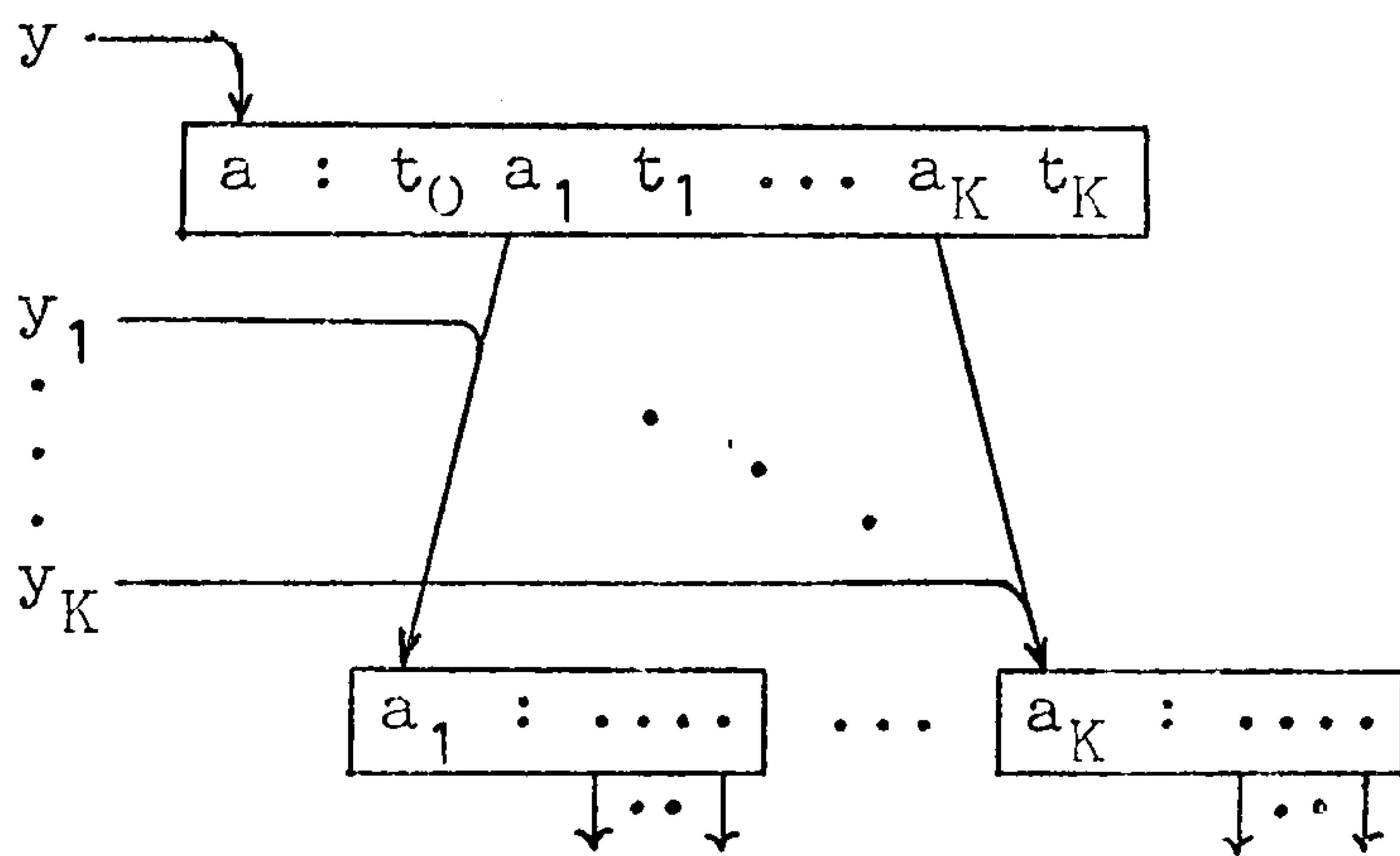
(1) trivially, if they are the self-same node;
or (2) if they are labelled by the same affix-rule, and all their direct descendants (if any) are, in order, pairwise equivalent.

Note that this one procedure implements all the equal-predicate functions. A flow diagram for the procedure is shown in figure 4.4, and some examples of pairs of trees which will be successfully matched by the procedure are shown in figure 4.5.

It may be seen that only a synthesise-predicate function can create a node and that the direct descendants of this node are fixed, once and for all, to be nodes which already exist. This guarantees that no loop can exist in the graph at any stage of a parse, and this in turn ensures that the procedure implementing the equal-predicate functions will terminate.



(a) Before application of the function.



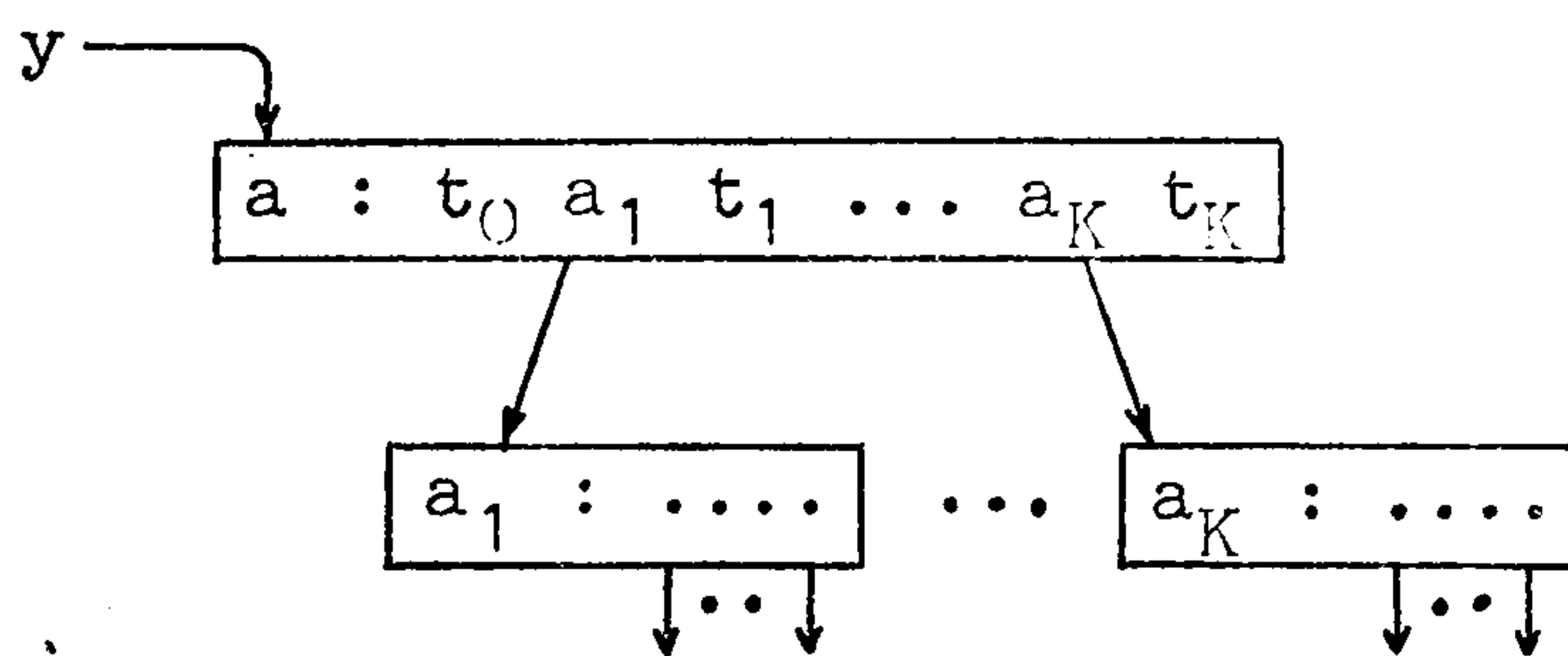
(b) After application of the function.

Figure 4.2 Action of the synthesise-predicate function

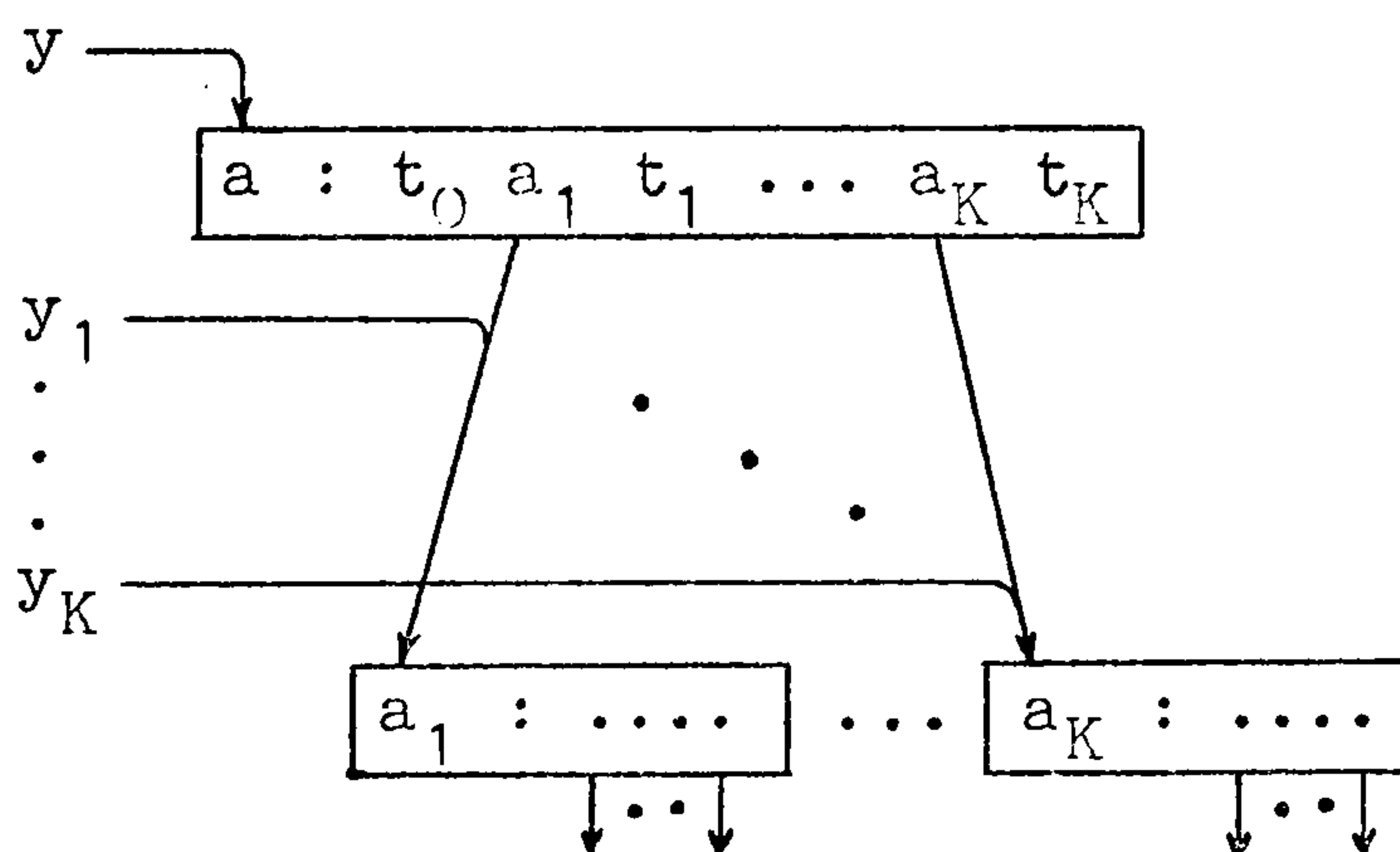
$$y = \tilde{F}_{\text{synth}_r}(y_1, \dots, y_K) ,$$

where affix-rule r is

$$a : t_0 a_1 t_1 \dots a_K t_K .$$



(a) Before application of the function.



(b) After application of the function .

Figure 4.3 Action of the analyse-predicate function

$$(y_1, \dots, y_K) = \tilde{F}_{\text{anal}_r}(y) ,$$

where affix-rule r is

$$a : \begin{matrix} \vdots \\ t_0 \end{matrix} a_1 \begin{matrix} \vdots \\ t_1 \end{matrix} \dots a_K \begin{matrix} \vdots \\ t_K \end{matrix} .$$

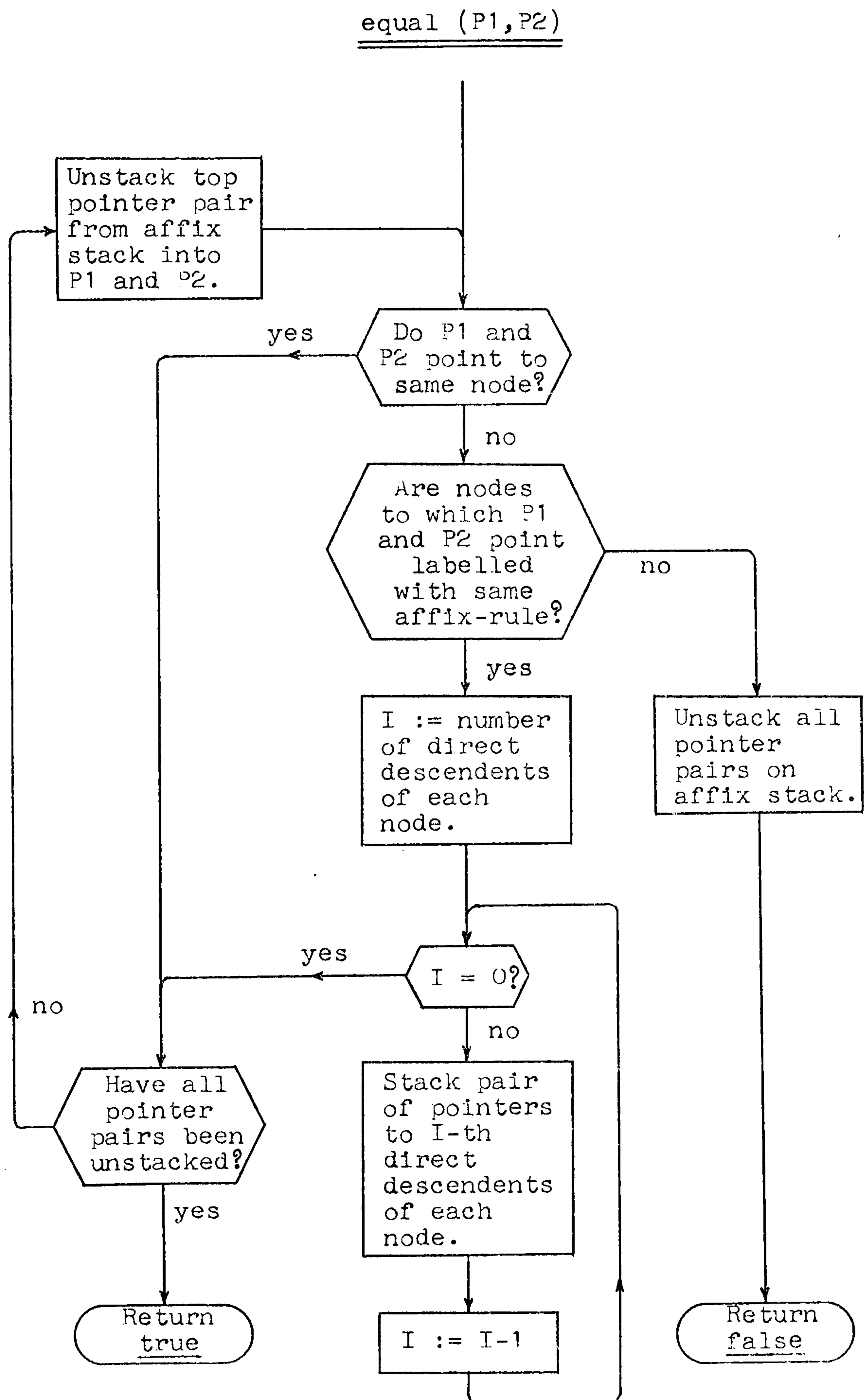
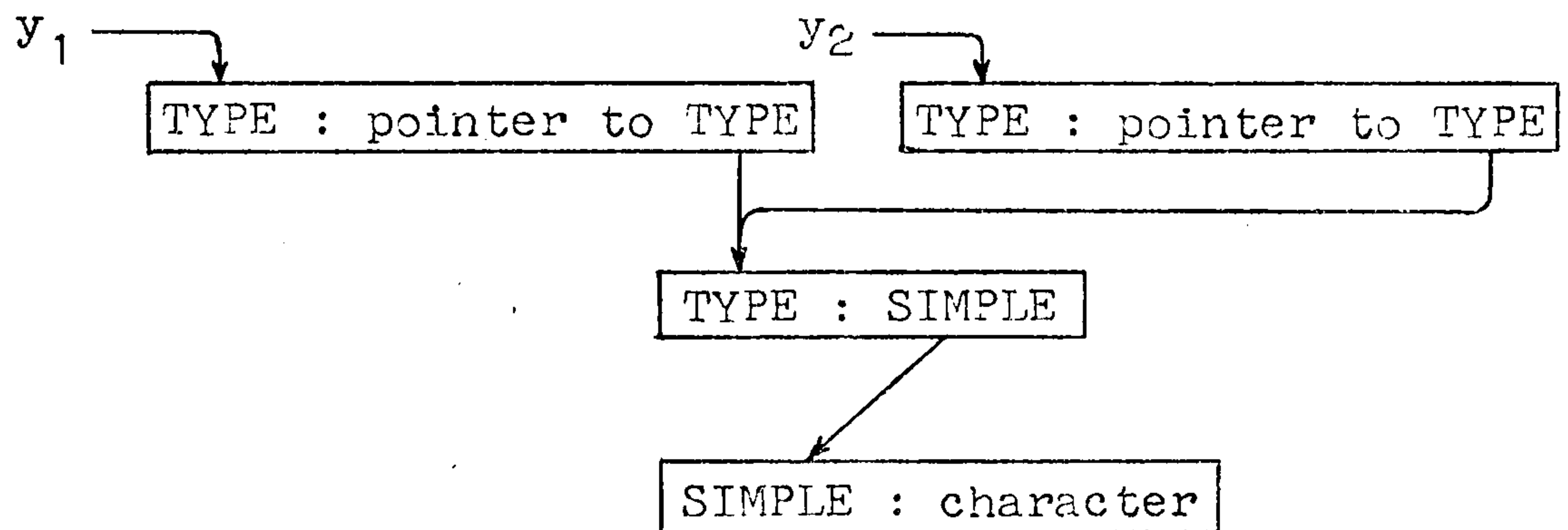


Figure 4.4 Flow diagram of the procedure used by the equal-predicate functions.



(a) $y_1 = y_2 = \text{'boolean'}$.



(b) $y_1 = y_2 = \text{'pointer to character'}$.

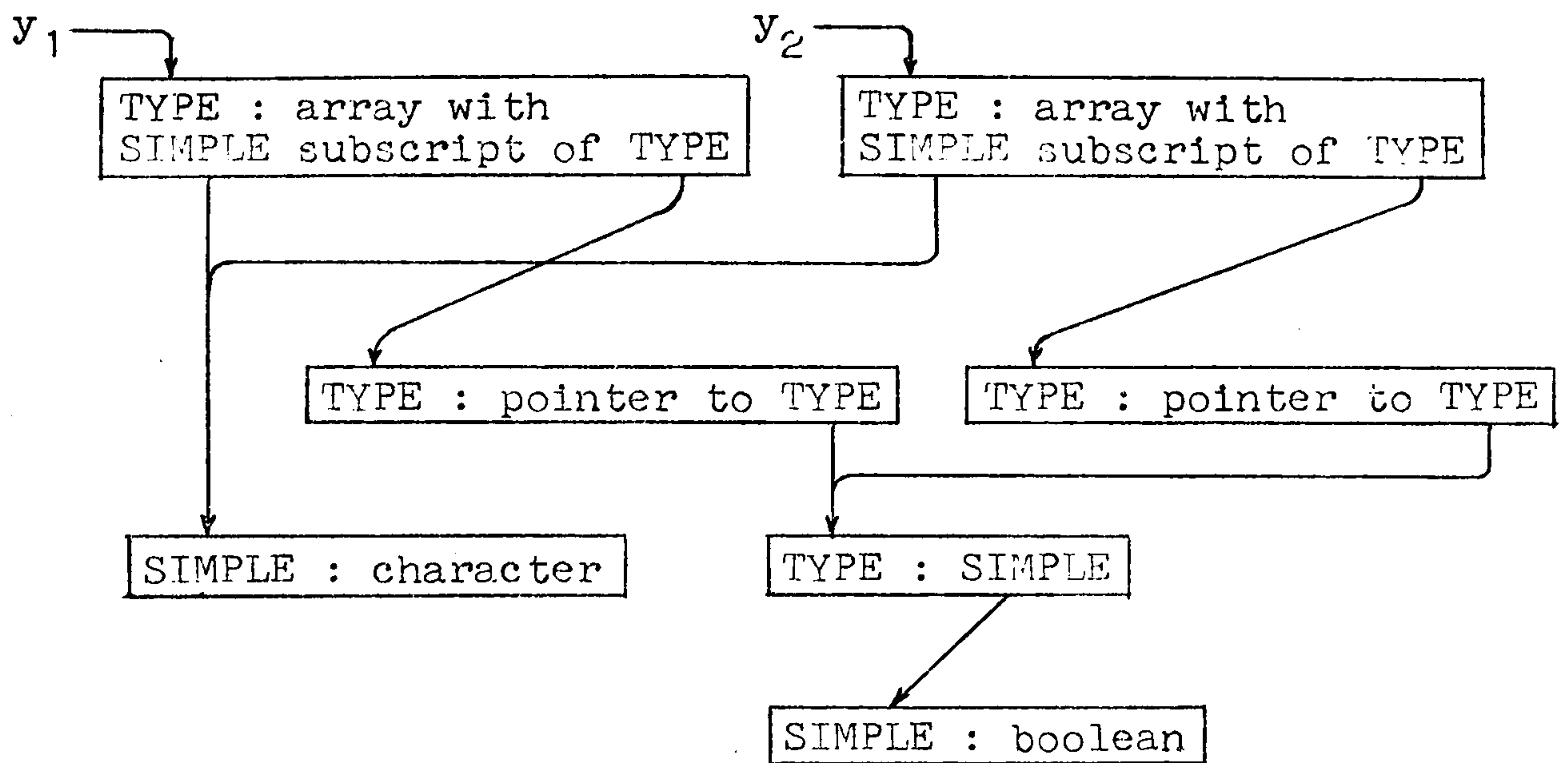
(continued)

Figure 4.5 Examples of derivation trees representing equal affixes.

The affix-rules are:-

TYPE : SIMPLE ; pointer to TYPE ;
array with SIMPLE subscript of TYPE .

SIMPLE : boolean ; character .



(c) $y_1 = y_2 =$ 'array with character subscript of pointer to boolean'.

Figure 4.5 (concluded)

Finally, it should be pointed out that our implementation of the primitive predicate functions does not require that the parser retain detailed knowledge of the affix-rules; it need know only the number, r , assigned to the affix-rule corresponding to each predicate synth_r or anal_r , and the number of affix-nonterminals in the right side of that affix-rule.

We are now in a position to show by example why our parser would malfunction if, in the AG constructed by our EAG-to-AG convertor, every affix-position of every hypernotation were not occupied by a variable whose associated affix-nonterminal specifies the domain of that affix-position. Figure 4.6(a) shows a fragment of an EAG and figure 4.6(b) the AG meta-rules constructed from it by our convertor. Figure 4.6(c) shows one of these meta-rules at an intermediate stage of the conversion; it is a well-formed AG meta-rule, since the affix-position of 'source' is inherited, and the domain of the variable 'SIMPLE' is a subset of the domain of the affix-position, namely $L(\text{TYPE})$. Suppose we allowed this meta-rule to stand, and suppose that in a particular parse both 'source' and 'expression' have the affix 'integer'. Then the variable 'SIMPLE' would be assigned a pointer to the derivation tree for

SIMPLE \Rightarrow integer .

This would be passed down as the inherited affix of 'source', and assigned to the variable 'TYPE' in the second meta-rule. The variable 'TYPE1' would be assigned the derived affix of 'expression', a pointer to the derivation tree for

TYPE \Rightarrow SIMPLE \Rightarrow integer .

The equal-predicate would then fail, as the two derivation trees are dissimilar, although the affixes they represent are the same.

4.3. Left Recursion in Extended Affix Grammars

We can investigate what left-recursive constructs in an EAG can be handled by our parsing method on the basis of our conclusions of section 2.8. At the same time we shall take the opportunity to introduce a useful optimisation into our EAG-to-AG convertor.

As EAG meta-rules do not contain primitive predicate hypernotations, we need concern ourselves only with those introduced by the convertor. Consider an EAG which contains the affix-rule

$$(r1) \quad C : d D$$

and the meta-rule

$$(p1) \quad v(dD) : v(dD) \dots ,$$

where v has one inherited affix-position whose domain is $L(C)$. Applying transformations T1 and T3 (section 3.4), we obtain the equivalent AG meta-rule

$$v(C) : \text{anal}_1(C,D) \text{synth}_1(D,C1) v(C1) \dots .$$

From section 2.8, the AG containing this delayed-left-recursive meta-rule cannot have an AF-LR(k) auxiliary grammar.

On both sides of p1, however, v 's affix-position is occupied by the same affix-form. It follows that, in every production rule generated from the meta-rule (and likewise in every production rule generated from the corresponding AG meta-rule), v 's affix will be the same on both sides. Thus an equivalent AG meta-rule can be constructed in which the same affix-variable is used on both sides, and in which the synthesise-predicate hypernotation is eliminated:

$$v(C) : \text{anal}_1(C,D) v(C) \dots .$$

Now there is nothing to prevent a re-ordering of the hypernotations to bring about a direct-left-recursive meta-rule:

$$v(C) : v(C) \text{anal}_1(C,D) \dots .$$

From section 2.8, a meta-rule of this form can easily be handled by our AF-LR(k) parsing technique.

We can generalise this result as follows. An EAG containing left-recursive meta-rules may be converted into an AG with an AF-LR(k) auxiliary grammar only if, in every left-recursive meta-rule, each inherited affix-position of the first hypernotation on the right side contains the same affix-form as the corresponding affix-position of the left-side hypernotation. All the nonterminals in an indirect left-recursive cycle must have inherited affix-positions with identical domains.

We have seen one example of the simplification of an AG meta-rule when the original EAG meta-rule contains identical affix-forms in two or more positions. Simplifications along these lines are always possible, during transformations T1-T4 (section 3.4), when identical affix-forms occur. We shall illustrate the simplifications with an EAG which contains the affix-rules

(r1) $C : d D$, (r2) $D : f F$,

and whose control includes (v, l, ι, C) , (w, l, ι, C) , and (x, l, δ, C) . There are three different cases.

(1) Two identical affix-forms both occur in applied positions. The right side of the unsimplified AG meta-rule will contain two similar sequences of synthesise-predicate hypernotations, introduced by transformations T1 or T4. The second of these sequences can be eliminated. For example:

EAG meta-rule:	$.. : \dots v(dfF) \dots w(dfF) \dots$
Unsimplified AG meta-rule:	$.. : \dots \text{synth}_2(F,D) \text{synth}_1(D,C)$ $v(C) \dots \text{synth}_2(F,D1)$ $\text{synth}_1(D1,C1) w(C1) \dots$
Simplified AG meta-rule:	$.. : \dots \text{synth}_2(F,D) \text{synth}_1(D,C)$ $v(C) \dots w(C) \dots$

(2) Two identical affix-forms occur, one in a defining position, one in an applied position. The right side of the unsimplified AG meta-rule will contain a sequence of synthesise-predicate hypernotations, introduced by transformations T1 or T4, defining the variable put into the applied position, and a sequence of analyse-

predicate hypernotations, introduced by transformations T2 or T3. Whichever sequence comes second can be eliminated. For example:

EAG meta-rule:	.. : ... x(dfF) ... v(dfF) ...
Unsimplified AG meta-rule:	.. : ... x(C) anal ₁ (C,D) anal ₂ (D,F) ... synth ₂ (F,D1) synth ₁ (D1,C1) v(C1) ...
Simplified AG meta-rule:	.. : ... x(C) anal ₁ (C,D) anal ₂ (D,F) ... v(C) ...

Our example involving left recursion came into this category.

(3) Two identical affix-forms both occur in defining positions. The right side of the unsimplified AG meta-rule will contain two similar sequences of analyse-predicate hypernotations, introduced by transformations T2 or T3. The second of these sequences can be eliminated. For example:

EAG meta-rule:	v(dfF) : ... x(dfF) ...
Unsimplified AG meta-rule:	v(C) : anal ₁ (C,D) anal ₂ (D,F) ... x(C1) anal ₁ (C1,D1) anal ₂ (D1,F) ...
Simplified AG meta-rule:	v(C) : anal ₁ (C,D) anal ₂ (D,F) ... x(C) ...

In this case, of course, an equal-predicate hypernotation will subsequently be introduced by transformation T5 or T6.

4.4. Multi-Predicate States in the Parser

In section 2.9, we showed how by a straightforward extension to the AF-LR(k) parsing algorithm, we could parse an auxiliary grammar whose AFSM contained multi-predicate states, provided that such states were deterministic. We defined disjointness of primitive predicate symbols and used it to state a sufficient condition for determinism of a multi-predicate state.

It is not, however, possible to decide whether two arbitrary primitive predicate symbols are disjoint. Consequently, it is not

Affix-rules:-

```
(r1)      MODE      :  TYPE formal value parameter ;
```

```
(r2)          TYPE formal variable parameter .
```

Nonterminals and control:-

actual-parameter	1	l	MODE
variable	1	s	TYPE
source	1	l	TYPE

Meta-rules:-

```
(p1)    actual-parameter(TYPE formal value parameter) :
        source(TYPE) .
```

```
(p2)      actual-parameter(TYPE formal variable parameter)
          : variable(TYPE) .
```

Figure 4.7 (a) A fragment of an EAG.

```
(p1) actual-parameter(MODE) : anal1(MODE,TYPE)
                                source(TYPE) .
```

```
(p2)      actual-parameter(MODE)  :  anal2(MODE,TYPE)
          variable(TYPE1) equalTYPE(TYPE1,TYPE) .
```

Figure 4.7 (b) AG meta-rules obtained from the above EAG meta-rules.

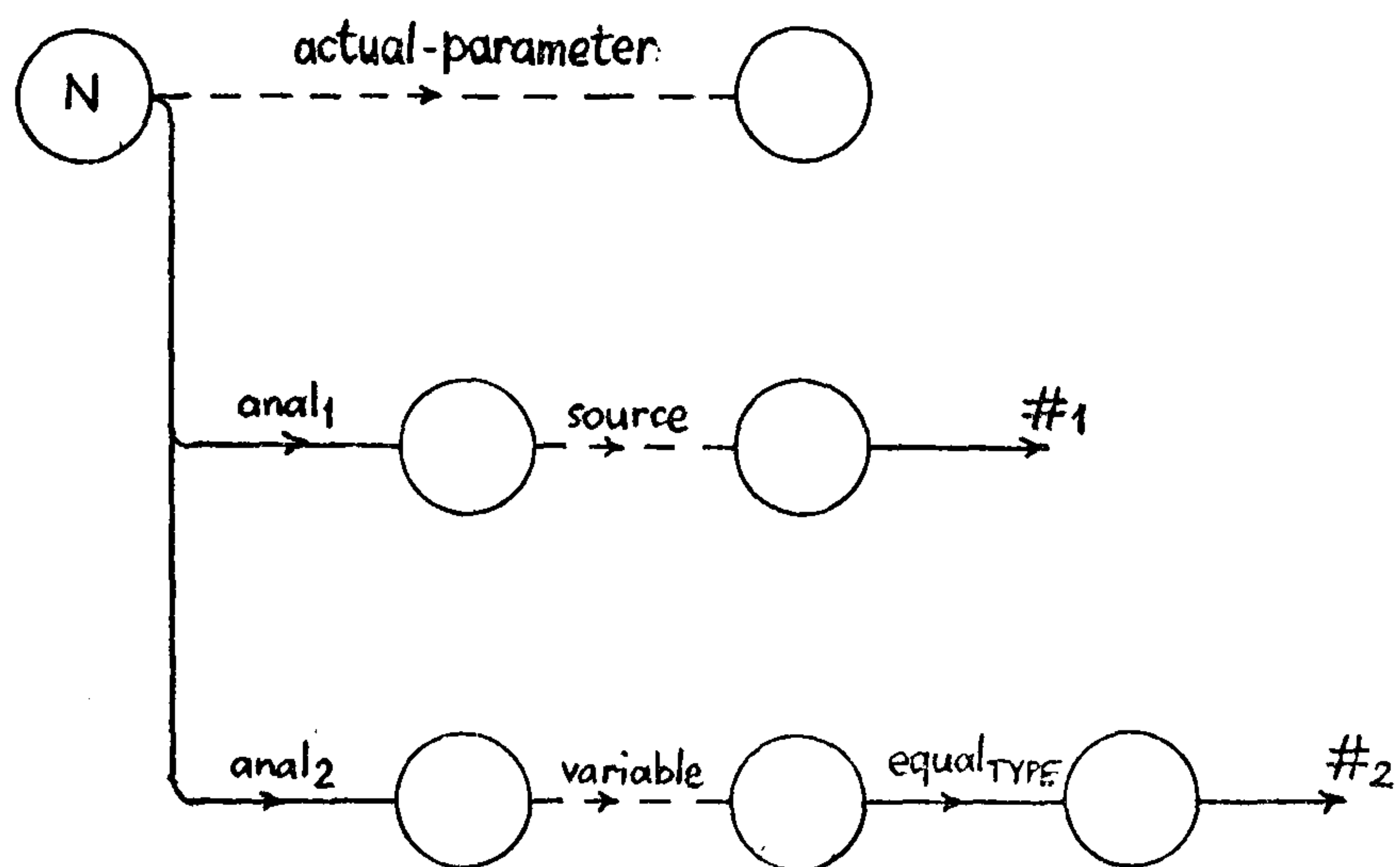


Figure 4.7(c) Part of an AFSM constructed from the above AG. State N is a deterministic multi-predicate state.

in general possible to determine automatically whether even our sufficient condition is satisfied.

Consider now the case of an AG constructed from an EAG. If a is a given affix-nonterminal, then all the primitive predicate symbols in the set $\{ \text{anal}_r \mid a \text{ is on the left side of affix-rule } r \}$ are mutually disjoint. Thus a multi-predicate state all of whose predicate-transitions are under analyse-predicates in such a set is deterministic.

Such multi-predicate states seem to occur quite often in practice, and they correspond to switches in ad-hoc parsers. Figure 4.7 illustrates how such a state can appear in an AFSM constructed from a practical grammar.

4.5. A Practical Implementation of the Parser

DeRemer (DeRemer 69) suggested a tabular representation for his LR(k) parsing automaton, and he described an interpreter which would be driven by the tables. In our AF-LR(k) parser, however, the actions to be performed in states of various types vary widely in complexity. For this reason it seems preferable to translate the AFSM into some sort of code than to adopt a tabular representation. The implementor, of course, still has a choice between the compactness of interpretive code and the speed of machine code.

In this section we describe a possible machine implementation of an AF-LR(k) parser constructed from an EAG. Since the extension is simple, we allow the parser to have multi-predicate states, provided that all the predicate-transitions from such states are under analyse-predicates.

Our implementation will require the following machine features:

- (1) 2 stacks - the "state stack" and the "affix stack";
- (2) a heap for storing the affix derivation trees;
- (3) a general purpose register, which we shall call X;
- (4) a buffer for storing up to k terminals.

For each state s there will be, in general, two code sequences. One code sequence, labelled Ns , will handle nonterminal transitions out of s . The other, labelled Ss , will handle all other transitions out of s . The state name of s on the state stack will be represented by the address of the code labelled Ns .

During a reduction, the nonterminal involved will be left in X , and control will be transferred to the label whose address is at the top of the state stack. The code labelled Ns will match X to one of the nonterminals which have transitions out of s , and transfer control accordingly (figure 4.8). Since this matching will always succeed, one of the nonterminal transitions is treated as a default case..

If there is only one nonterminal transition out of s , to s_1 say, then the address of Ss_1 , instead of Ns , can be used as the state name of s .

If there is no nonterminal transition out of s , then the address of Ns will never be at the top of the state stack after a reduction, and therefore an arbitrary value can be used for the state name of s . (That is, when this state name is to be stacked, it is necessary only to adjust the stack pointer.)

The larger part of the parser body will be composed of code sequences implementing terminal-, reduce-, predicate- and copy-transitions. These code sequences are shown in figure 4.9. Note that a reduce state name is not stacked, since it would immediately be unstacked, unless the right-side head string of the meta-rule involved is empty (see figure 4.9(b) and (c)).

The adjustment of affixes in the affix stack in reduce states (figure 4.9(b) and (c)) amounts simply to a re-ordering or copying of items near the top of the affix stack. The necessary code (which is not shown in figure 4.9) is determined by the auxiliary grammar meta-rule involved. For example, if the meta-rule is

$$\begin{pmatrix} \text{vble} \\ L \ M \end{pmatrix} : \begin{pmatrix} \text{tag identify} \\ L \ T \quad M \end{pmatrix} ,$$

then the top 3 affixes will be matched to the variables L, T and M, and these are to be replaced by the affixes matched to L and M. Thus the necessary code is simply

"copy top item in affix stack into 2nd top position in affix stack", which will be followed by code to decrement the affix stack pointer by 1. Quite frequently, as in the case of the meta-rule

$$\begin{pmatrix} \text{stmt} \\ L \end{pmatrix} : \begin{pmatrix} \text{vble} := \text{v7 vble} \text{ v8 equal_MODE} \\ L \ M \quad L \ M1 \ M1 \ M \end{pmatrix} ,$$

no re-ordering code at all is necessary. Sometimes the affix stack pointer will not be altered, sometimes it will even be incremented.

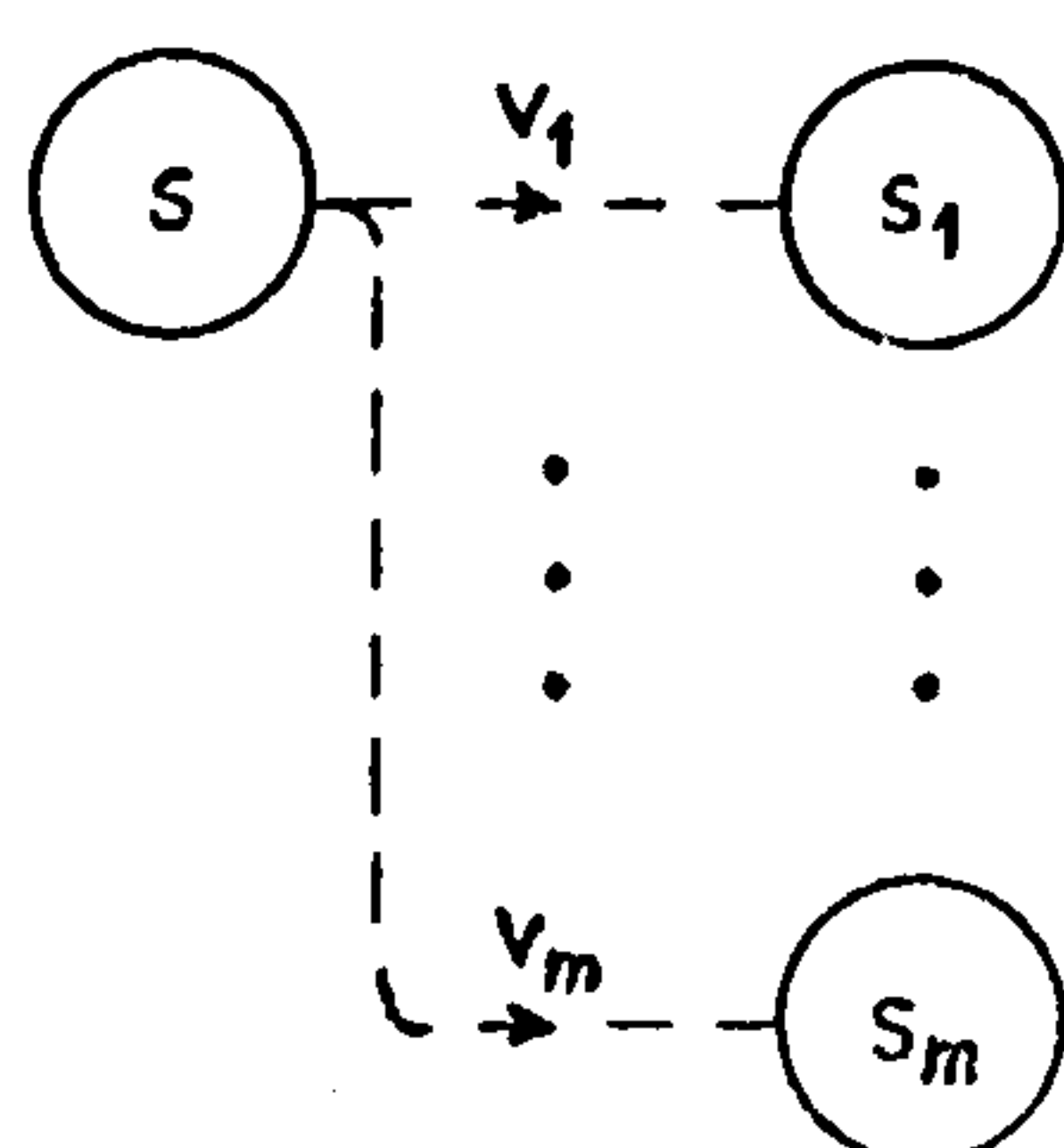
Similar code will be used in copy states, although in this case each step will simply copy an item in the affix stack to the top of the stack. For example, if the auxiliary grammar meta-rule involved is

$$\begin{pmatrix} \text{v8} \\ M \ L \ M1 \ M1 \ M \end{pmatrix} : \begin{pmatrix} \\ M \ L \ M1 \end{pmatrix} ,$$

then, assuming that each "Stack" operation automatically increments the stack pointer by 1, the necessary code is

"stack top item in affix stack on affix stack;
stack 4th top item in affix stack on affix stack".

We have seen several possibilities for local code optimisation. There are also situations where code optimisation of a more global nature is possible. These possibilities also arise in ordinary LR(k) parsers (DeRemer 69).



Ns: If X contains v_1 , jump to Ss_1 ;

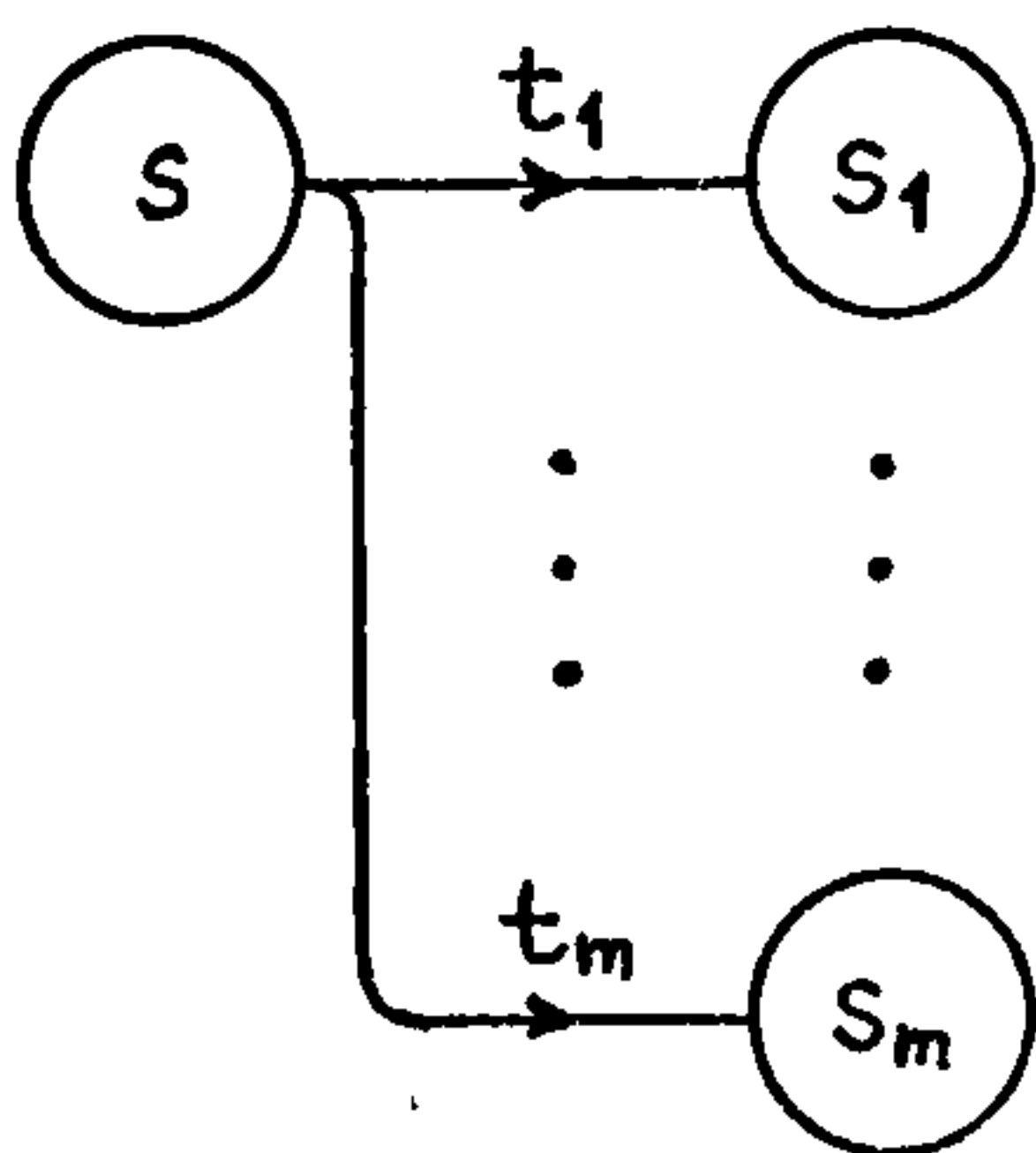
·
·
·

if X contains v_{m-1} , jump to Ss_{m-1} ;

jump to Ss_m .

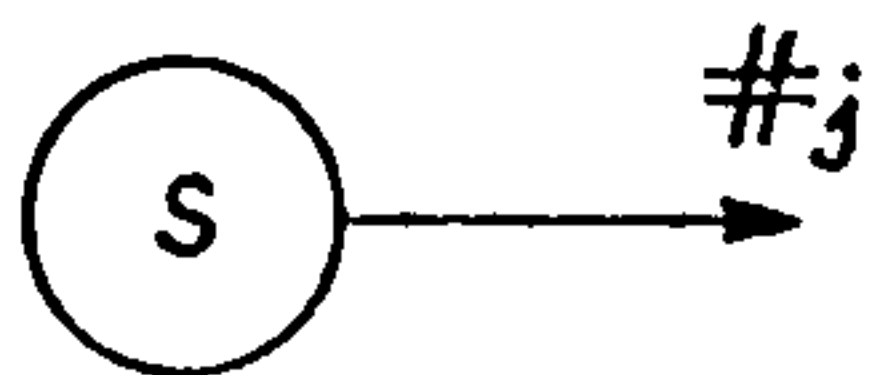
Figure 4.8 Code sequence implementing nonterminal transitions out of a state (of any type).

(Only nonterminal transitions are shown in the diagram.)



```
Ss: Stack address of Ns on state stack;
    read next terminal into X;
    if X contains  $t_1$ , jump to  $Ss_1$ ;
    .
    .
    .
    if X contains  $t_m$ , jump to  $Ss_m$ ;
    jump to error routine.
```

Figure 4.9 (a) Code sequence implementing terminal transitions out of a read state.



Auxiliary grammar
meta-rule j is

$$\begin{pmatrix} v \\ k \end{pmatrix} : \begin{pmatrix} \rho \\ \theta \end{pmatrix}$$

and n = length of ρ ,
 m = length of θ ,
 l = length of κ .

```

Ss: Record reduction;

    adjust affixes in affix stack;
                                /* if necessary */

    pop (m-1) affixes;          /* if m≠1 */

    pop (n-1) states;           /* if n>1 */

    put v in register X;

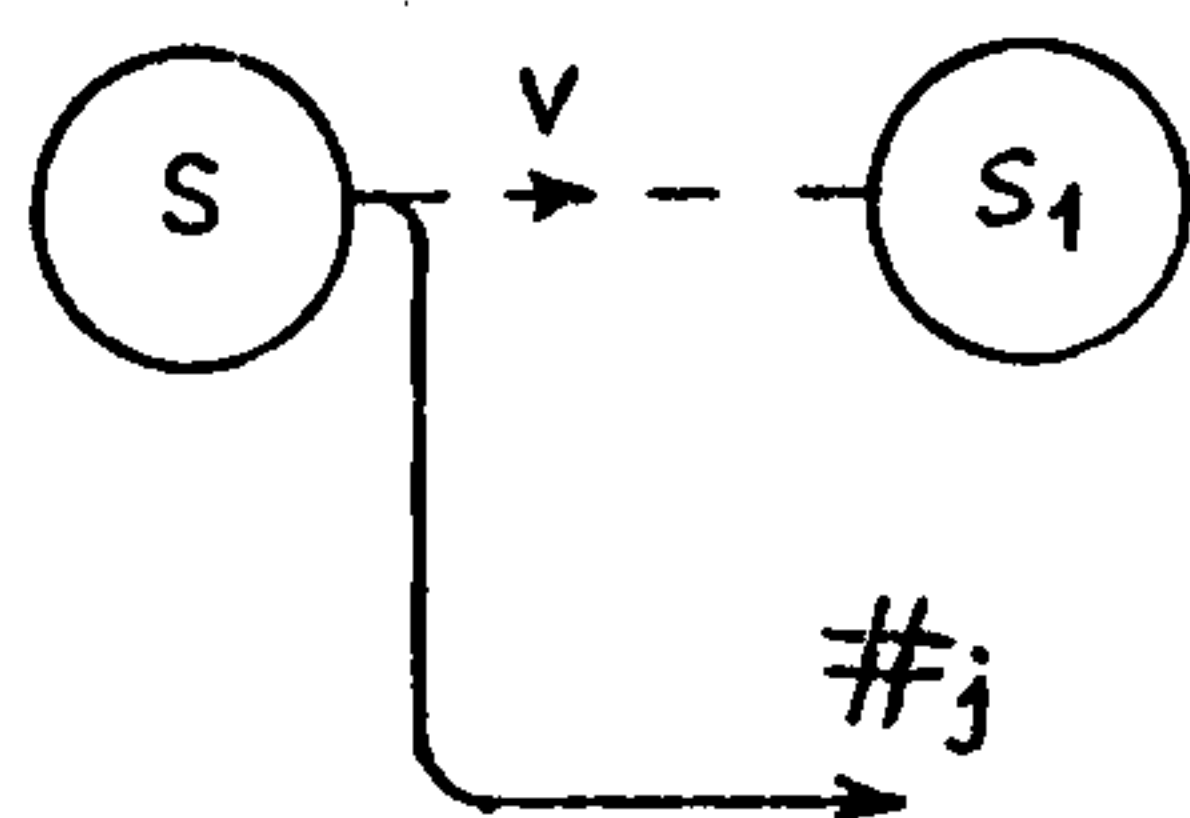
    jump to label whose address is at top
        of state stack.

```

Figure 4.9 (b) Code sequence implementing the reduce transition out of a reduce state, where the right-side head string of the auxiliary grammar meta-rule involved is non-empty.

If $v = e$, replace the last two lines of code by

```
"stop - parse successful."
```



Auxiliary grammar
meta-rule j is

$$\begin{pmatrix} v \\ \kappa \end{pmatrix} : \begin{pmatrix} \lambda \\ \theta \end{pmatrix}$$

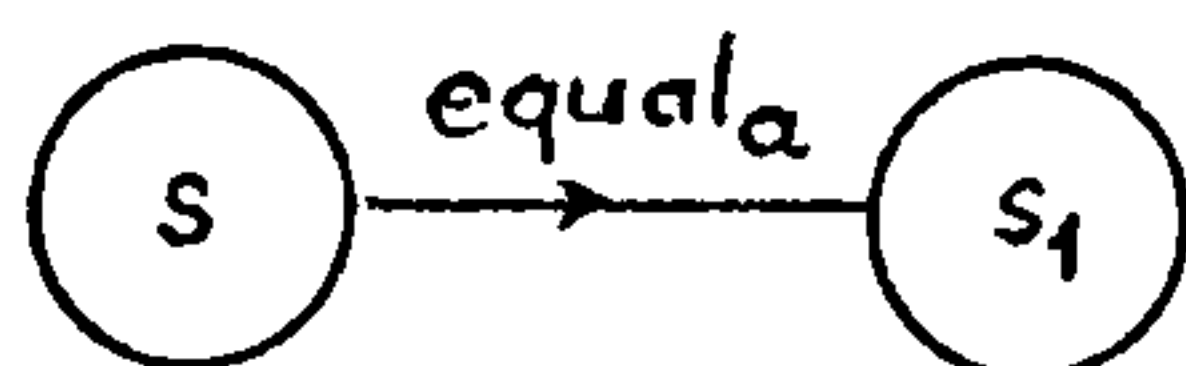
and m = length of θ ,
 l = length of κ .

Ss: Stack address of N_s on state stack;
 record reduction;
 adjust affixes in affix stack;
 /* if necessary */
 pop $(m-1)$ affixes; /* if $m \neq 1$ */
 jump to Ss_1 .

Figure 4.9 (c) Code sequence implementing the reduce transition out of a reduce state, where the right-side head string of the auxiliary grammar meta-rule involved is empty.

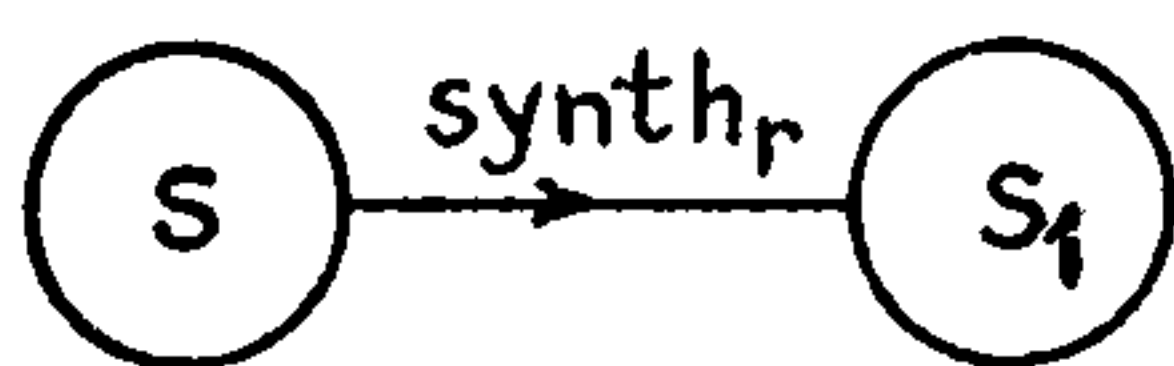
If $v = e$, replace the last two lines of code by

"stop - parse successful."



Ss: Stack address of N_s on state stack;
 call equal-predicate function with top two affixes in affix stack as parameters, and on failure jump to error routine;
 jump to Ss_1 .

Figure 4.9 (d) Code sequence implementing an equal-predicate transition out of a predicate state.



```
Ss: Stack address of Ns in state stack;
    createnew node, with n+2 fields, in
                                heap;
```

```
put r in field 1 of node;
```

```
put n in field 2 of node;
```

```

copy n-th top item in affix stack
into field 3 of node;

```

•

```

copy 1st top item in affix stack
into field (n+2) of node;

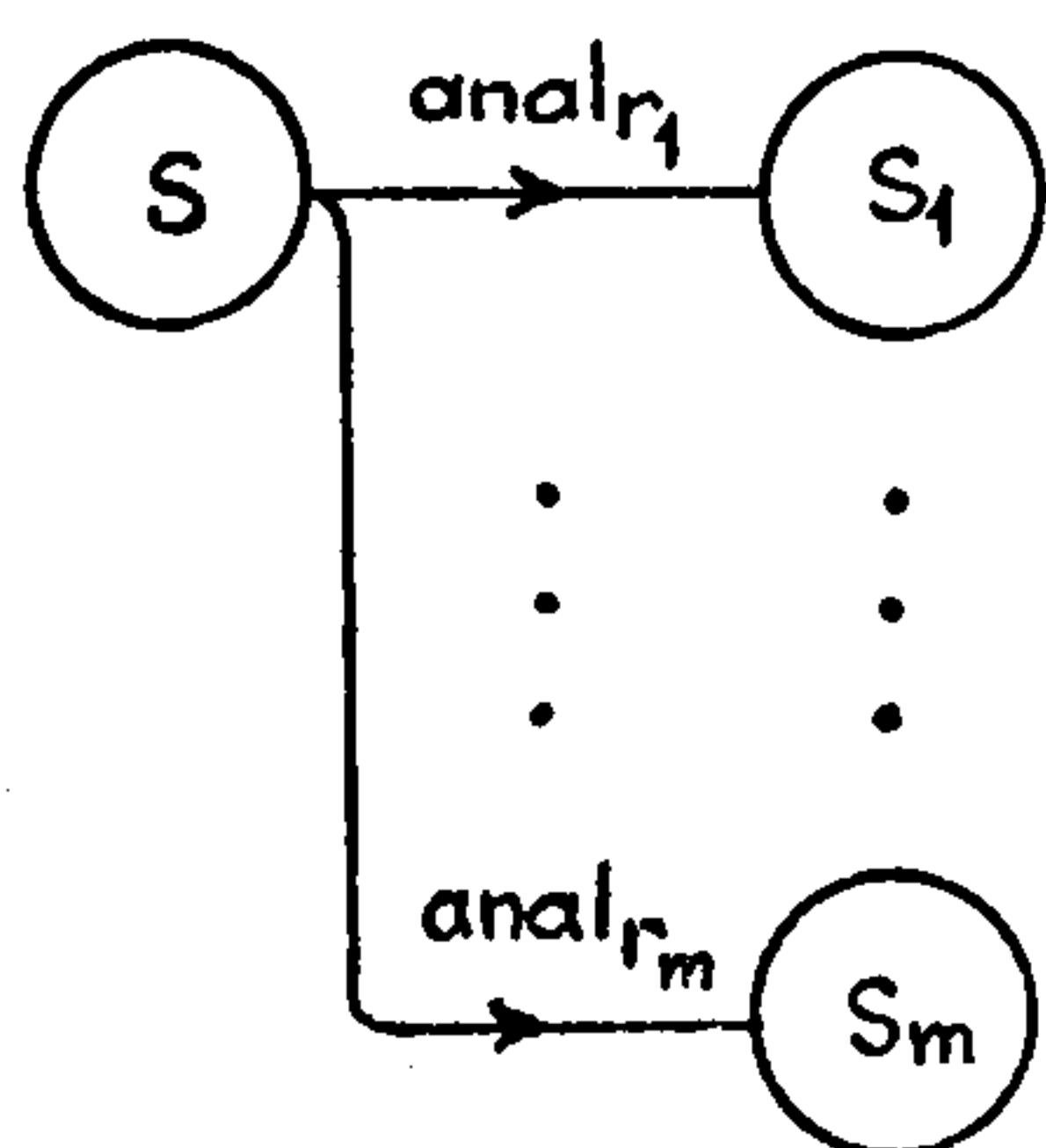
```

```
stack pointer to node on affix stack;
```

jump to Ss_1 .

Affix-rule r has
 n affix-nonterminals
on its right side.

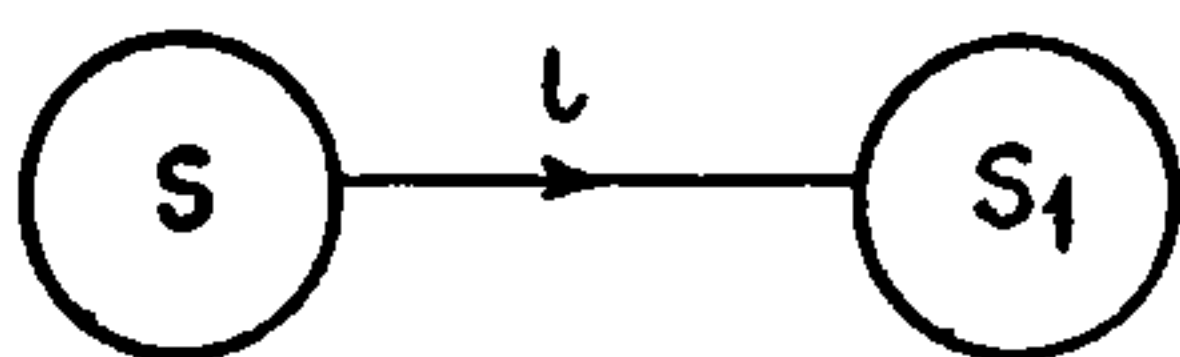
Figure 4.9 (e) Code sequence implementing a synthesise-predicate transition out of a predicate state.



For each $i = 1, \dots, m$,
 affix-rule r_i
 has n_i affix-
 nonterminals
 on its right
 side.

Ss: Stack address of N_s in state stack;
 fetch into register X field 1 of node
 to which top item in affix stack
 points;
 if X does not contain r_1 , jump to sL2;
 stack field 3 of node on affix stack;
 .
 .
 .
 stack field (n_1+2) of node on affix
 stack;
 jump to Ss_1 .
 sL2:
 .
 .
 .
 sLm: if X does not contain r_m , jump to
 error routine;
 stack field 3 of node on affix stack;
 .
 .
 .
 stack field (n_m+2) of node on affix
 stack;
 jump to Ss_m .

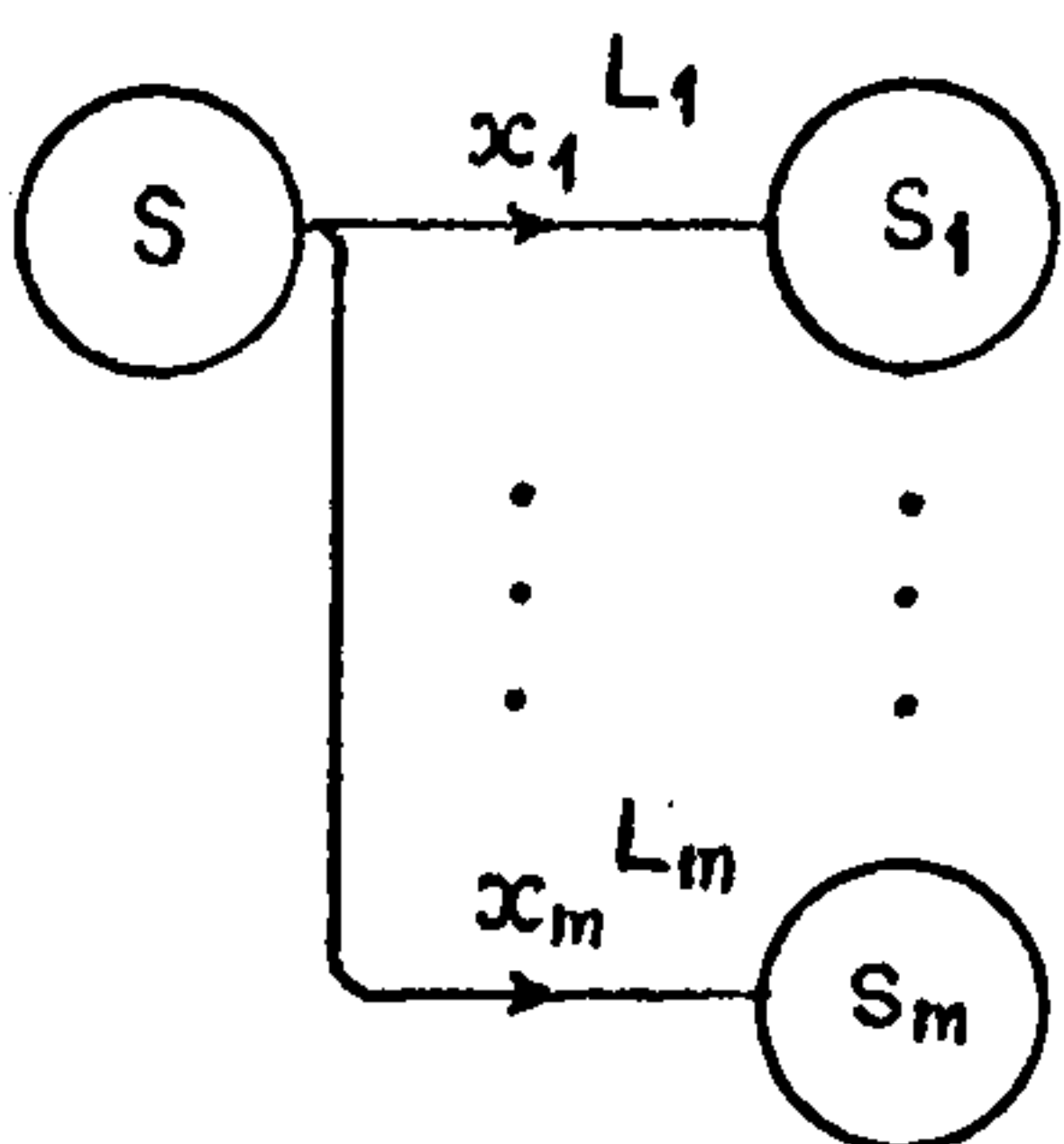
Figure 4.9 (f) Code sequence implementing analyse-predicate transition(s) out of a (multi-)predicate state.



Ss: Stack address of Ns on state stack;
 copy affix(es) to top of affix stack;
 jump to Ss₁.

Figure 4.9 (g)

Code sequence implementing the copy-transition out of a copy state.



Ss: Stack address of Ns on state stack;
 bring next k terminals into buffer;
 if buffer string is not in set L₁,
 jump to sL2;
 <code for transition under x₁>

sL2:

⋮

sLm: if buffer string is not in set L_m,
 jump to error routine;

<code for transition under x_m>

L₁, ..., L_m
 are k-symbol
 look-ahead
 sets.

Figure 4.9 (h)

Code sequence implementing transitions out of a k-look-ahead state.

<code for transition under x₁> depends on whether x₁ is a terminal, #-symbol, copy-symbol, or primitive predicate symbol. If x₁ is a terminal, the code sequence is

"read next terminal into X; jump to Ss₁."

Otherwise the code sequence is similar to that in (b), (c), (d), (e), (f) or (g) above.

It often happens that there is only one state to which control can be transferred from a given reduce state. In such a case the general method of transferring control from a reduce state, using the label address at the top of the state stack and the nonterminal involved, can be replaced by a direct jump. Figure 4.9 (c) shows one simple case. Another example can be found in figure 2.10: control can be transferred directly from state 24 to state 9. The reason for this is that the head grammar production rule involved is

$$\text{stmts} \rightarrow \text{stmts} ; \text{stmt}$$

and the only path to state 24 which spells out 'stmts ; stmt' starts from state 4, so state 4 will always be at the top of the state stack when control leaves state 24; and the transition under 'stmts' from state 4 goes to state 9. In general, control can be transferred directly from reduce state R to state M only if

- (1) the head grammar production rule associated with the #-transition out of R is $N \rightarrow v$;
 - and (2) for every state Q such that v spells out a path from Q to R, the transition under N out of Q goes to M.
- All this applies equally to #-transitions out of look-ahead states.

Frequently, several read states may have in common a subset of transitions under the same terminals to the same destination states. A good example of this may be seen in figure 2.10: the transitions under 'x', 'y' and 'z' from states 2, 4, 18 and 25. Such a subset of transitions should be translated into a single code sequence which is shared by all the states concerned. The same applies to nonterminal transitions out of states of any type. In practice the parser body can be compacted considerably in this way.

4.6. Optimisations to the Parser Implementation

In describing our implementation of the EAG parser in section 4.5, we mentioned some code optimisations, most of which were equally relevant to an LR(k) parser for a CFG. In this section we consider other optimisations which arise out of the construction of the parser from an EAG.

When a primitive predicate symbol occurs in an auxiliary grammar meta-rule, it may be preceded by a copy-symbol. The function of the copy-symbol at parse time is to copy to the top of the affix stack the inherited affixes of the primitive predicate. (The copy-symbol will be absent only when these affixes are guaranteed to be at the top of the stack already.) Now, we have chosen to translate each predicate-transition into open code. Thus it is unnecessary for the inherited affixes to be at the top of the stack: the open code could easily be modified to access the affixes below the top of the stack.

Consider, for example, the auxiliary grammar meta-rules

$$\begin{aligned} \left(\begin{array}{c} \text{identify} \\ L1 \ T \ M \end{array} \right) &: \left(\begin{array}{c} \text{!l} \ \text{anal}_7 \ \text{!2} \ \text{identify} \\ L1 \ T \ L1 \ L \ T1 \ M1 \ L \ T \ M \end{array} \right) , \\ \left(\begin{array}{c} \text{!l} \\ L1 \ T \ L1 \end{array} \right) &: \left(\begin{array}{c} \text{!l} \\ L1 \ T \end{array} \right) , \end{aligned}$$

which would be constructed from meta-rule pll of figure 3.3.

'!l' causes the 2nd top item in the affix stack to be copied to the top, where it is examined by the 'anal₇' function. The code sequences for the consecutive transitions under '!l' and 'anal₇' could be merged into the following sequence:-

```
"Ss: Stack address of Ns in state stack;
    fetch into register X field 1 of node to which 2nd top item
        in affix stack points;
    if X does not contain 7, jump to error routine;
    stack field 3 of node on affix stack;
    stack field 4 of node on affix stack;
    stack field 5 of node on affix stack;
    jump to Ss1."
```

Of course, the meta-rule must be altered accordingly; this can be achieved by a suitable alteration to the algorithm which converts

an AG meta-rule into an optimised auxiliary grammar meta-rule (section 2.10). In our example, the meta-rule would be changed to

$$\left(\begin{array}{c} \text{identify} \\ L1 \ T \ M \end{array} \right) : \left(\begin{array}{c} \text{!l-anal}_7 \ \text{!2 identify} \\ L1 \ T \ L \ T1 \ M1 \ L \ T \ M \end{array} \right) ,$$

where ' !l-anal_7 ' is a new symbol conveying the meaning that the inherited affix of (this particular instance of) ' anal_7 ' will be found in a stack location which can be deduced from the meta-rule for ' !l '.

Figure 4.10 shows an example of an auxiliary grammar optimised in this way.

Our second optimisation concerns the representation of affixes. There are certain special classes of affixes, readily determinable from the EAG, for which more efficient representations than trees are possible.

Suppose a is an affix-nonterminal, and the affix-rules which have a on their left sides are

$$a : t_1 ; t_2 ; \dots ; t_m ,$$

where each t_i is a string of affix-terminals. Then each affix t_i in $L(a)$ could be represented by the number r_i of the affix-rule $a : t_i$ (using the same numbering of affix-rules as in the construction of section 3.4). The synthesise- and analyse-predicate functions would then be as follows:-

$$\begin{aligned} \tilde{F}_{\text{synth}_r} &= r && \text{(a 0-parameter function, i.e. a constant),} \\ \tilde{F}_{\text{anal}_r} &= \lambda n \ (\ n=r \mid \phi \mid \omega \) && . \end{aligned}$$

Examples of affixes which could be handled in this way are the terminal productions of 'TAG' and of 'MODE' in the EAG of figure 3.1.

If a is an affix-nonterminal with two affix-rules of the form

$$\begin{aligned} (r1) \quad a &: t ; \\ (r2) \quad &s a u , \end{aligned}$$

where t , s and u are strings of affix-terminals, then each affix in $L(a)$ is of the form ' $s^n t u^n$ ' (where $n \geq 0$) and could be represented by the integer n . The synthesise- and analyse-predicate functions would then be arithmetic:-

$$\begin{aligned}\tilde{F}_{\text{synth}_1} &= 0, \\ \tilde{F}_{\text{anal}_1} &= \lambda n (n=0 \mid \phi \mid \omega), \\ \tilde{F}_{\text{synth}_2} &= \lambda n (n+1), \\ \tilde{F}_{\text{anal}_2} &= \lambda n (n>0 \mid n-1 \mid \omega).\end{aligned}$$

All these functions could be implemented very easily, as could the equal-predicate functions $\tilde{F}_{\text{equal}_a}$, which would simply test two integers for equality. The only difficulty arises when pointer and integer representations for different affixes co-exist: both pointers and integers may occupy the "direct-descendant" fields of a node. The general equal-predicate function (figure 4.4) therefore must be able to distinguish between the two representations, and must be modified to take appropriate action when it encounters an integer. For example, on a machine in which addresses are positive integers, negative integers could be used in the integer representation.

4.7. Efficiency of the Parser Implementation

In this section we describe a simple empirical investigation into the efficiency of a parser constructed from an EAG. For this purpose we chose a language which is small enough to be easily manageable but which contains enough of the features of a typical programming language for the results of the experiment to be realistic. Our language is a small subset of PASCAL (Wirth 71, 73), and features an infinity of data types and assignments between variables declared to be of identical type.

The EAG defining our language is shown in figure 4.11. (Actually, we have cheated a little: this EAG includes one primitive predicate symbol, 'uneqtag', which has the property

$I = \{ \iota_1, \iota_2, \iota_3, \iota_4, \iota_5, \iota_6, \iota_7 \}$

$P_A :-$

- (p1) $(\text{program}) : \left(\begin{array}{c} \text{begin} \\ \text{L} \end{array} \text{synth}_8 \text{synth}_1 \text{synth}_5 \text{synth}_7 \right.$
 $\quad \quad \quad \text{synth}_2 \text{synth}_4 \text{synth}_7 \text{synth}_3 \text{synth}_5$
 $\quad \quad \quad \text{synth}_6 \text{ } \iota_1\text{-synth}_7 \text{stmts } \text{end} \left. \right)$.
- (p2) $(\text{stmts}) : (\text{stmt}) ;$
 $\quad \quad \quad \text{L} \quad \quad \quad \text{L}$
- (p3) $(\text{stmts} ; \text{stmt})$.
 $\quad \quad \quad \text{L}$
- (p4) $(\text{stmt}) : \left(\begin{array}{c} \text{vble} := \iota_2 \text{vble } \iota_3\text{-equal} \\ \text{L M} \quad \quad \quad \text{L M1} \end{array} \text{MODE} \right)$.
- (p5) $(\text{vble}) : \left(\begin{array}{c} \text{tag identify} \\ \text{L T} \quad \quad \quad \text{M} \end{array} \right) ;$
- (p6) $(\text{vble anal}_6 [\iota_4 \text{vble anal}_4])$.
 $\quad \quad \quad \text{L M1} \quad \quad \quad \text{M} \quad \quad \quad \text{L M2}$
- (p7) $(\text{tag}) : \left(\begin{array}{c} x \text{synth}_1 \\ \text{T} \end{array} \right) ;$
- (p8) $(y \text{synth}_2)$;
 $\quad \quad \quad \text{T}$
- (p9) $(z \text{synth}_3)$.
 $\quad \quad \quad \text{T}$
- (p10) $(\text{identify}) : \left(\begin{array}{c} \iota_5\text{-anal}_7 \iota_6\text{-equal} \\ \text{L1 T L T1 M} \end{array} \text{TAG} \right)$.
- (p11) $(\begin{array}{c} \iota_5\text{-anal}_7 \iota_7 \text{identify} \\ \text{L1 T L T1 M1 L T M} \end{array})$.

(continued)

Figure 4.10. Auxiliary grammar constructed from the AG of figure 3.3 using the optimisation described in section 4.6 merging copy- and predicate-symbols.

$$\begin{aligned}
(p12) \quad & \left(\begin{array}{cccccc} & & & \iota^1 & & \\ L2 & T2 & M2 & M3 & L2 & T2 & M3 \end{array} \right) : \left(L2 \ T2 \ M2 \ M3 \right) \cdot \\
(p13) \quad & \left(\begin{array}{ccc} \iota^2 & & \\ L & M & L \end{array} \right) : \left(L \ M \right) \cdot \\
(p14) \quad & \left(\begin{array}{ccccc} & & \iota^3 & & \\ M & L & M1 & M1 & M \end{array} \right) : \left(M \ L \ M1 \right) \cdot \\
(p15) \quad & \left(\begin{array}{cccc} \iota^4 & & & \\ L & M1 & M & L \end{array} \right) : \left(L \ M1 \ M \right) \cdot \\
(p16) \quad & \left(\begin{array}{ccc} \iota^5 & & \\ L1 & T & L1 \end{array} \right) : \left(L1 \ T \right) \cdot \\
(p17) \quad & \left(\begin{array}{cccccc} & & \iota^6 & & & \\ T & L & T1 & M & T1 & T \end{array} \right) : \left(T \ L \ T1 \ M \right) \cdot \\
(p18) \quad & \left(\begin{array}{cccccc} & & \iota^7 & & & \\ T & L & T1 & M1 & L & T \end{array} \right) : \left(T \ L \ T1 \ M1 \right) \cdot
\end{aligned}$$

Figure 4.10 (concluded)

of being disjoint (section 2.9) from the symbol 'equal_{TAG}'.) An AFSM was generated from the EAG using our automatic constructor, and the AFSM was manually translated into KDF9 Usercode (ICL-) according to the implementation method described in sections 4.5 and 4.6. The parser was AF-LALR(1), apart from a few multi-predicate states.

To provide a meaningful basis for comparison, a CFG defining the context-free features of our language (see figure 4.12) was used to construct a CFSM, which was also translated manually into KDF9 Usercode, using a similar implementation method. This parser was LALR(1). The code was then modified to impose the context-sensitive constraints on the language, for example identification and type compatibility. This additional code was tailored to the language requirements, except that no attempt was made to exploit the small number of different identifier tags, for example by using table look-up for identification. List structures were used for a tag-type table and to represent types.

(KDF9 Usercode is the assembly language of the KDF9, a 48-bit machine with a $1\mu s$ cycle time and $6\mu s$ store and with hardware operand and return-address stacks of 16 and 15 registers respectively. No attempt was made to exploit the unconventional features of the machine's architecture in the implementation of either parser, lest the results of the experiment be untypical.)

For test purposes a set of 7 programs was used; these are listed in figure 4.13. Of these, programs 1 - 4 were chosen at random, and programs 5 - 7 were designed to be similar to one another in structure, so as to reveal any underlying features (such as linearity) in the performance of the parsers.

The results of the experiment are presented in figure 4.14. Each parser was applied to each test program, and the parsing time and the heap size were measured. The number of reductions performed by the parsers were also noted. The storage requirements of the parser bodies themselves and their service routines are recorded

in the table. The timing results are presented graphically in figure 4.15, which shows plots of parsing time against program length, and in figure 4.16, which shows plots of parsing time against program length plus number of reductions. The latter plots reflect the larger number of reductions performed by the EAG parser, due to the more complete formalisation of the language definition by the EAG.

The results show that the EAG parser was slower than the CFG-parser-based analyser by a factor of about 2. The performance of the CFG parser was, as expected, linear. The performance of the EAG parser showed a nonlinear component if only the program length were considered, but this was not so apparent if the number of reductions were also taken into account.

The principal reason for the relative slowness of the EAG parser was probably the inefficiency of the identification process. The EAG meta-rules pl6 - pl8 effectively define a recursive procedure, which is equivalent to, but much slower than, the linear search of the ad-hoc identification routine invoked by the CFG parser. In addition there were the overheads of additional stack operations, notably the repeated copying of the pointer to the tag-type list. In the CFG-parser-based analyser, the corresponding pointer was stored in a global location.

It must be emphasised, however, that such a simple ad-hoc solution might not be available in a programming language more complex than our miniature language. This language was exceptional in that, in EAG terms, many of the notions occurring in a parse have one and the same inherited affix (a terminal production of 'LIST') - the one which was repeatedly copied in the affix stack. In general, such copying cannot easily be avoided.

Affix-rules:-

(r1-r4) TAG : w ; x ; y ; z .
 (r5) TYPE : SIMPLE ;
 (r6) array with SIMPLE subscript of TYPE .
 (r7-r8) SIMPLE : integer ; boolean .
 (r9-r10) TYPEQ : TYPE ; undefined .
 (r11) LIST : ;
 (r12) LIST TAG TYPE .

Control:-

block				
declns	1	δ	LIST	
stmts	1	ι	LIST	
stmt	1	ι	LIST	
variable	2	(ι, δ)	(LIST, TYPE)	
tag	1	δ	TAG	
type	1	δ	TYPE	
local	3	(ι, ι, δ)	(LIST, TAG, TYPEQ)	
uneqtag	2	(ι, ι)	(TAG, TAG)	$\lambda t \lambda u (t \neq u)$

(continued)

Figure 4.11 The EAG used in the experiment described in section 4.7.

Meta-rules:-

- (p1) block : var declns(LIST) begin stmts(LIST) end .
- (p2) declns() : .
- (p3) declns(LIST TAG TYPE) : declns(LIST) tag(TAG) :
 type(TYPE) ; local(LIST,TAG,undefined) :
- (p4) stmts(LIST) : stmt(LIST) ;
- (p5) stmts(LIST) ; stmt(LIST) .
- (p6) stmt(LIST) : variable(LIST,TYPE) :=
 variable(LIST,TYPE) .
- (p7) variable(LIST,TYPE) : tag(TAG)
 local(LIST,TAG,TYPE) ;
- (p8) variable(LIST,array with SIMPLE subscript
 of TYPE) [variable(LIST,SIMPLE)] .
- (p9) tag(w) : w .
- (p10) tag(x) : x .
- (p11) tag(y) : y .
- (p12) tag(z) : z .
- (p13) type(integer) : integer .
- (p14) type(boolean) : boolean .
- (p15) type(array with SIMPLE subscript of TYPE) :
 array [type(SIMPLE)] of type(TYPE) .
- (p16) local(LIST TAG TYPE, TAG, TYPE) : .
- (p17) local(LIST TAG1 TYPE1, TAG, TYPEQ) :
 uneqtag(TAG1,TAG) local(LIST, TAG, TYPEQ) .
- (p18) local(, TAG, undefined) : .

Figure 4.11 (concluded)

The terminals ':' and ';' are underlined in the meta-rules to distinguish them from the EAG metasymbols.

(p1)	block	->	<u>var</u> declns <u>begin</u> stmts <u>end</u>
(p2)	declns	->	
(p3)	declns	->	declns tag <u>:</u> type <u>;</u>
(p4)	stmts	->	stmt
(p5)	stmts	->	stmts <u>;</u> stmt
(p6)	stmt	->	variable := variable
(p7)	variable	->	tag
(p8)	variable	->	variable [variable]
(p9)	tag	->	w
(p10)	tag	->	x
(p11)	tag	->	y
(p12)	tag	->	z
(p13)	type	->	integer
(p14)	type	->	boolean
(p15)	type	->	<u>array</u> [type] <u>of</u> type

Figure 4.12 The CFG used in the experiment described in section 4.7.

Program 1

```
var x : integer ; y : integer ;  
begin y := x end
```

Program 2

```
var w : array [ boolean ] of integer ;  
      z : boolean ; y : integer ;  
begin y := w [ z ] ; w [ z ] := y end
```

Program 3

```
var w : integer ;  
      y : array [ integer ] of array [ boolean ]  
          of boolean ;  
      z : array [ boolean ] of boolean ;  
      x : boolean ;  
begin y [ w ] := z ;  
      z [ z [ x ] ] := y [ w ] [ x ]  
end
```

Program 4

```
var x : integer ; y : boolean ;  
      z : array [ boolean ] of integer ;  
      w : array [ integer ] of array [ boolean ]  
          of boolean ;  
begin x := z [ y ] ;  
      z [ y ] := x ;  
      w [ x ] [ y ] := y ;  
      y := w [ z [ y ] ] [ y ]  
end
```

(continued)

Figure 4.13 Test programs generated by the grammars of figures 4.11 and 4.12.

Program 5

```

var z : integer ;
      x : array [ integer ] of integer ;
begin z := x [ x [ x [ x [ x [ x [ x [ x [
              z ] ] ] ] ] ] ] ]
end

```

Program 6

```

var z : integer ;
      x : array [ integer ] of integer ;
begin z := x [ x [ x [ x [ x [ x [ x [ x [
              x [ x [ x [ x [ x [ x [ x [ x [
              z ] ] ] ] ] ] ] ] ] ] ] ] ] ] ] ] ] ]
end

```

Program 7

```

var z : integer ;
      x : array [ integer ] of integer ;
begin z := x [ x [ x [ x [ x [ x [ x [ x [
              x [ x [ x [ x [ x [ x [ x [ x [
              x [ x [ x [ x [ x [ x [ x [ x [
              z ] ] ] ] ] ] ] ] ] ] ] ] ] ] ] ] ] ]
              ] ] ] ] ] ] ] ] ] ] ] ] ] ] ] ] ] ]
end

```

Figure 4.13 (concluded)

	EAG Parser			CFG Parser based Analyser		
Size of code:-						
Parser body		249			152	
Service routines		26			19	
Timings, etc.:-						
Program	Length	Time (ms)	Heap	Red'ns	Time (ms)	Heap Red'ns
1	14	7.0	28	20	3.4	6 14
2	33	18.5	57	49	7.8	11 31
3	56	31.7	90	79	13.3	18 47
4	70	46.0	108	116	17.3	18 65
5	43	21.1	59	55	9.8	8 40
6	67	34.4	83	87	15.4	8 64
7	91	46.7	107	119	21.0	8 88

Figure 4.14 Results of an empirical investigation into the efficiencies of parsers constructed from the EAG of figure 4.11 and the CFG of figure 4.12. For the test programs used, see figure 4.13.

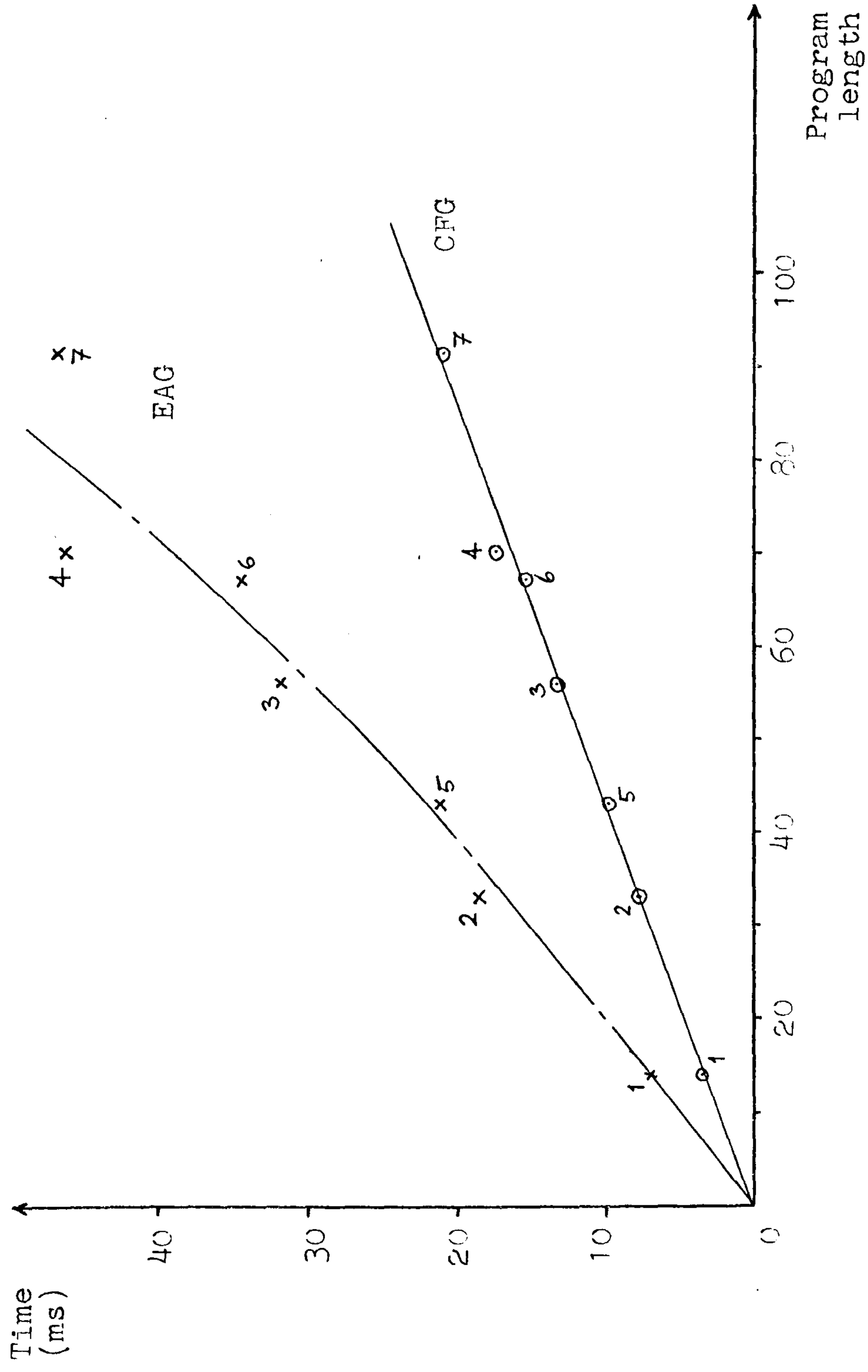


Figure 4.15 Plot of parsing time against program length for an EAG parser and a CFG-parser-based analyser (taken from the results quoted in figure 4.14).

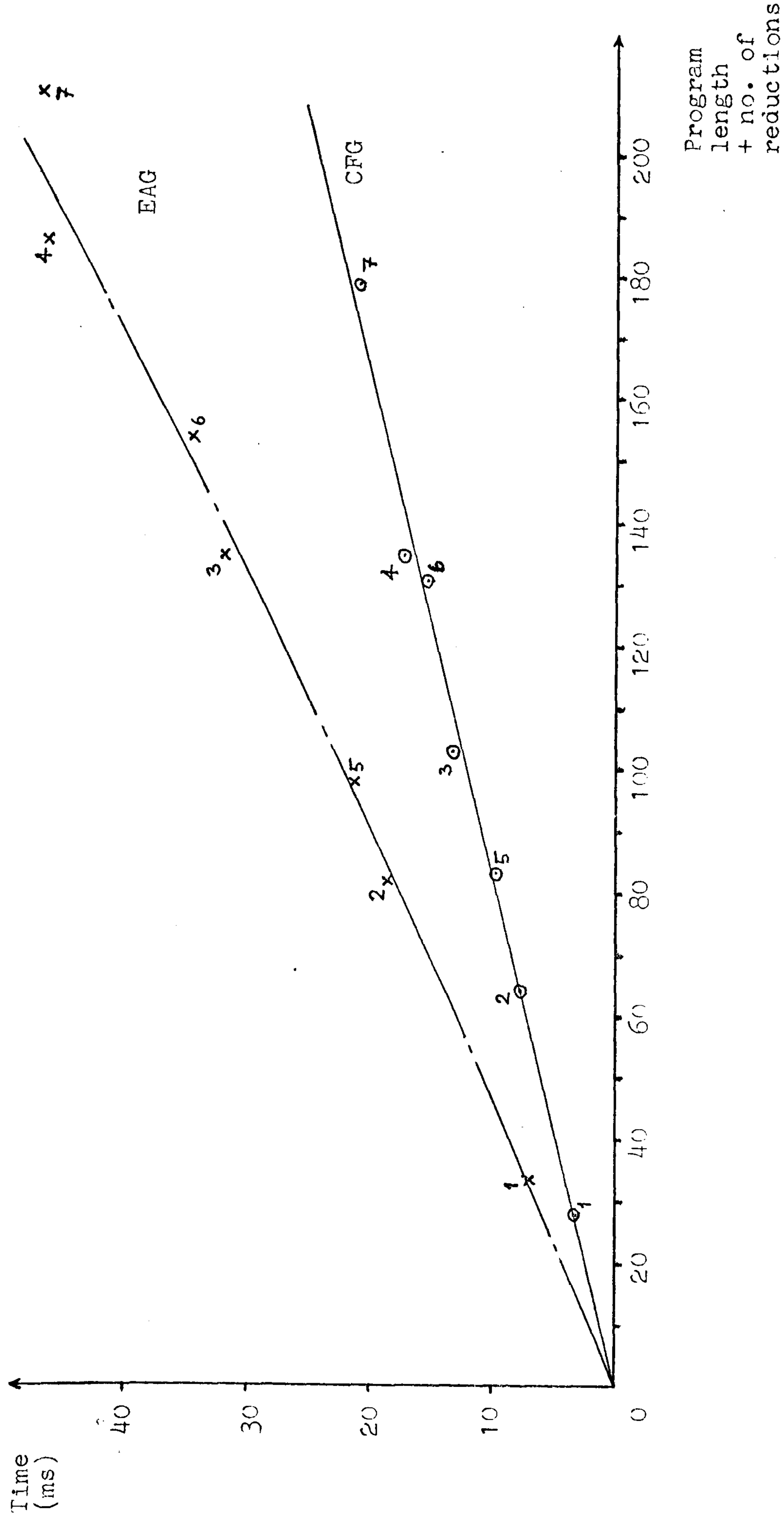


Figure 4.16 Plot of parsing time against (program length + number of reductions) for an EAG parser and a CFG-parser-based analyser (taken from the results quoted in figure 4.14).

4.8. Incorporation of the Parser into a Translator

The experiment of section 4.7 demonstrated that our EAG parser is not quite efficient enough as it stands to be a practical parser. Further optimisations could possibly cut down some of the overheads, but the principal inefficiency was the identification process, that is the parsing of notions whose head is 'identify'. Now, as it happens, every such notion either has one terminal production, the empty string, or none at all; and in the former case the inherited affixes are effectively mapped on to the derived affix. Thus this particular nonterminal performs a role which would be performed by a primitive predicate symbol in a well-formed AG. (See figure 1.1.)

A gain in efficiency, at the expense of rigour, could be achieved by treating a symbol such as 'identify' as a primitive predicate symbol and removing those meta-rules which have that symbol on their left sides. Meta-rules which contain that symbol in their right sides need not be altered in any way. Our constructor could easily handle primitive predicate symbols in EAG meta-rules. It would be necessary to designate to the constructor which symbols are to be treated in this way, and to program separately the functions mapping their inherited affixes on to their derived affixes. It is useful, however, that a rigorous and binding specification of these mappings exists in the meta-rules which were removed. Thus the strict original EAG retains its definitive role.

Another aspect of the incorporation of our parser in a practical translator is its interface with a lexical analyser. A lexical analyser in a practical programming language translator not only recognises terminal symbols but also analyses objects such as identifier tags and constants and passes them to the parser as single symbols rather than sequences of terminals. In practice this approach reduces the size and increases the speed of the parser, and makes the grammar more likely to be LR(k).

In a strict EAG, such objects are likely to correspond to

nonterminals with affixes (usually derived). If the lexical analyser is to treat them as terminals, we must informally allow terminals to have affixes in an EAG. Again, our constructor can easily handle this possibility. As before, the EAG meta-rules need not be altered, except for the removal of those which have these new "terminals" on their left sides. Of course, the lexical analyser must stack the affixes when such a "terminal" is read by the parser.

In the case of identifier tags, each could have an affix represented by an integer, which could be determined by a hashing function. This suffices for distinguishing between different tags, and the composition of the tag as a sequence of letters and digits has no inherent significance.

A practical example of this approach to the incorporation of our parser in a compiler appears in the appendix (section A.2).

A practical EAG parser which uses a tree representation for some of its affixes will require a garbage collector. Since no structure containing loops can be created by our parser, a straightforward reference-counting method can be used to detect inaccessible nodes. Moreover, the nature of our parser's tree manipulations enable us to predict to some extent how the reference counts will vary. We assume that each node has an additional field for its reference count.

It is simplest to think in terms of the intermediate AG meta-rules. First of all, we note that multiple copies of a pointer on the affix stack can be counted as one. If an affix-variable occurs on the right side, but not on the left side, of an AG meta-rule, then on a reduction involving that meta-rule the reference count of the node to which the value of that variable points must be decreased by one. All other reference count adjustments are connected with the evaluation of primitive predicate functions. In the case of an analyse-predicate function,

the nodes to which the (pointers representing the) derived affixes point have their reference counts increased by one; in the case of a synthesise-predicate function, the nodes to which the inherited affixes point have their reference counts increased by one, and the node to which the derived affix points has its reference count initialised to one.

Apart from the initialisation of the reference counts of newly created nodes, all reference count adjustments should be delayed until reduction time. Often adjustments to a particular node will cancel out. For example, in meta-rule p3 of figure 3.3, the nodes to which the values of L, T and M point would have their reference counts incremented by the 'synth₇' function and decremented again by the reduction itself (since they do not occur on the left side), so no adjustments at all to these nodes are necessary.

When the reference count of a node is decreased, it will be necessary to check whether it has become zero. If so, the node will be released to the pool of free space. Any direct descendants of the node will have their own reference counts decreased by one, and of course it is possible that some of them will also be released as a result.

CHAPTER 5

CONCLUSIONS

5.1. Applications of Extended Affix Grammars

We have already emphasised the application of EAGs to language definition, and referred to the practical example of such an application given in Appendix A.

We do not claim that every programming language can be naturally defined by a well-formed EAG. Indeed, we should expect any well-formed EAG defining ALGOL 68 to bear little resemblance, in parts, to the definitive syntax in (van Wijngaarden 68)! Although not very suitable for parsing, non-well-formed EAGs may well have a part to play in language definition.

Another possible application of EAGs is translation. Consider the EAG fragment shown in figure 5.1. Each instance of a notion whose head is 'expression' will have a derived affix which is a postfix representation of an expression derived from it. Still more to the point, our representation of that affix (section 4.1) will be, in effect, an operator tree representation of the expression. These analogies are illustrated in figure 5.2.

Probably any desired tree representation of a sentence can conveniently be modelled by suitable affix-rules, and the translation of the sentence into such a representation can be defined by the meta-rules of the EAG.

We suspect that application of EAGs along these lines may

Affix-rules:-

```
OBJ  : OBJ OBJ add ;
      OBJ OBJ multiply ;
      OBJ negate ;
      ID .
```

```
ID  : a ; b ; c ; ...
```

Control:-

expression	1	δ	OBJ
term	1	δ	OBJ
factor	1	δ	OBJ
identifier	1	δ	ID

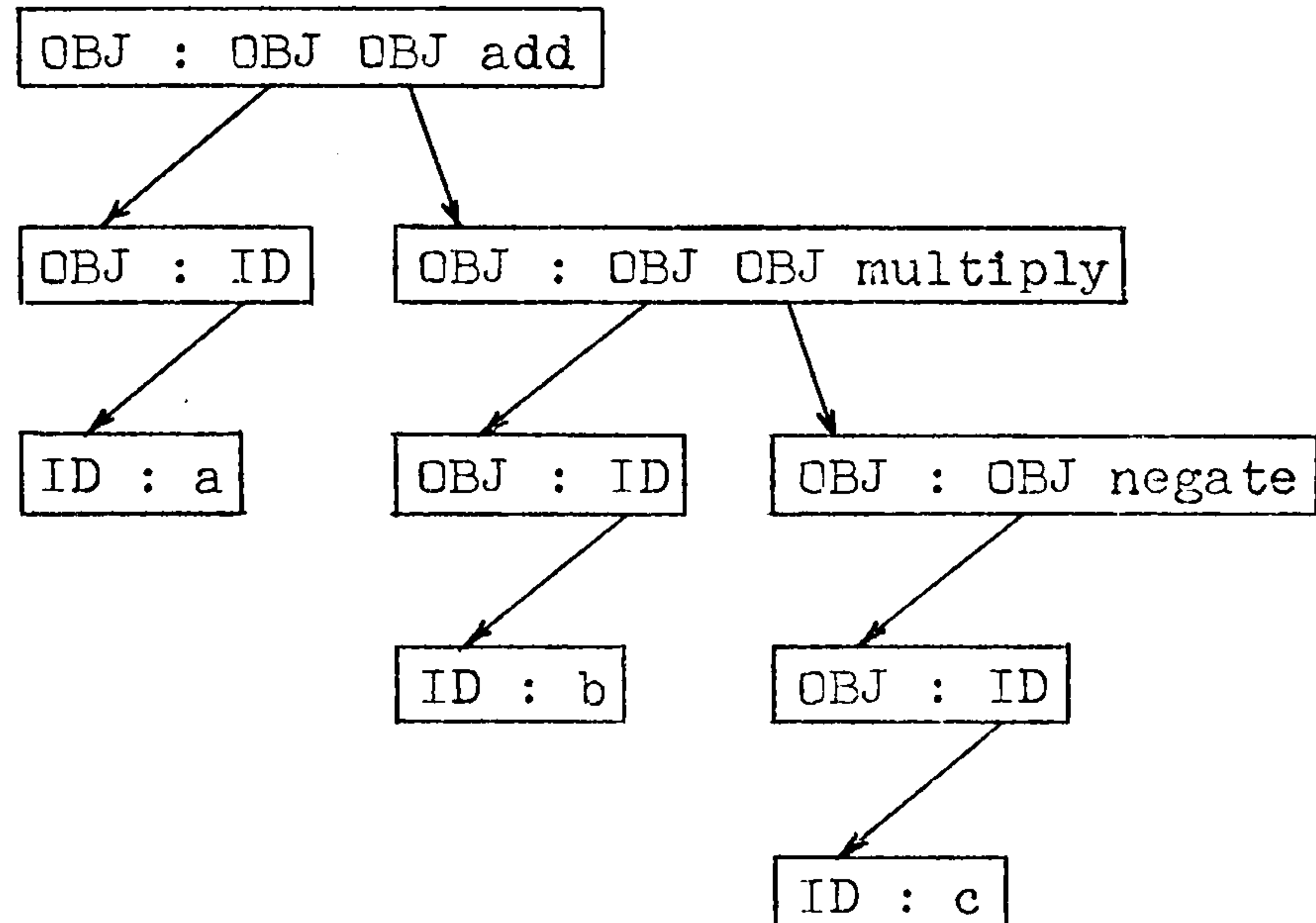
Meta-rules:-

```
expression(OBJ) : term(OBJ) .
expression(OBJ negate) : - term(OBJ) .
expression(OBJ1 OBJ2 add) : expression(OBJ1) +
                             term(OBJ2) .
term(OBJ) : factor(OBJ) .
term(OBJ1 OBJ2 multiply) : term(OBJ1) *
                             factor(OBJ2) .
factor(ID) : identifier(ID) .
factor(OBJ) : ( expression(OBJ) ) .
```

Figure 5.1 Fragment of an EAG defining translation of an arithmetic expression into postfix (or tree) form.

(a) a b c negate multiply add

(b)



(c)

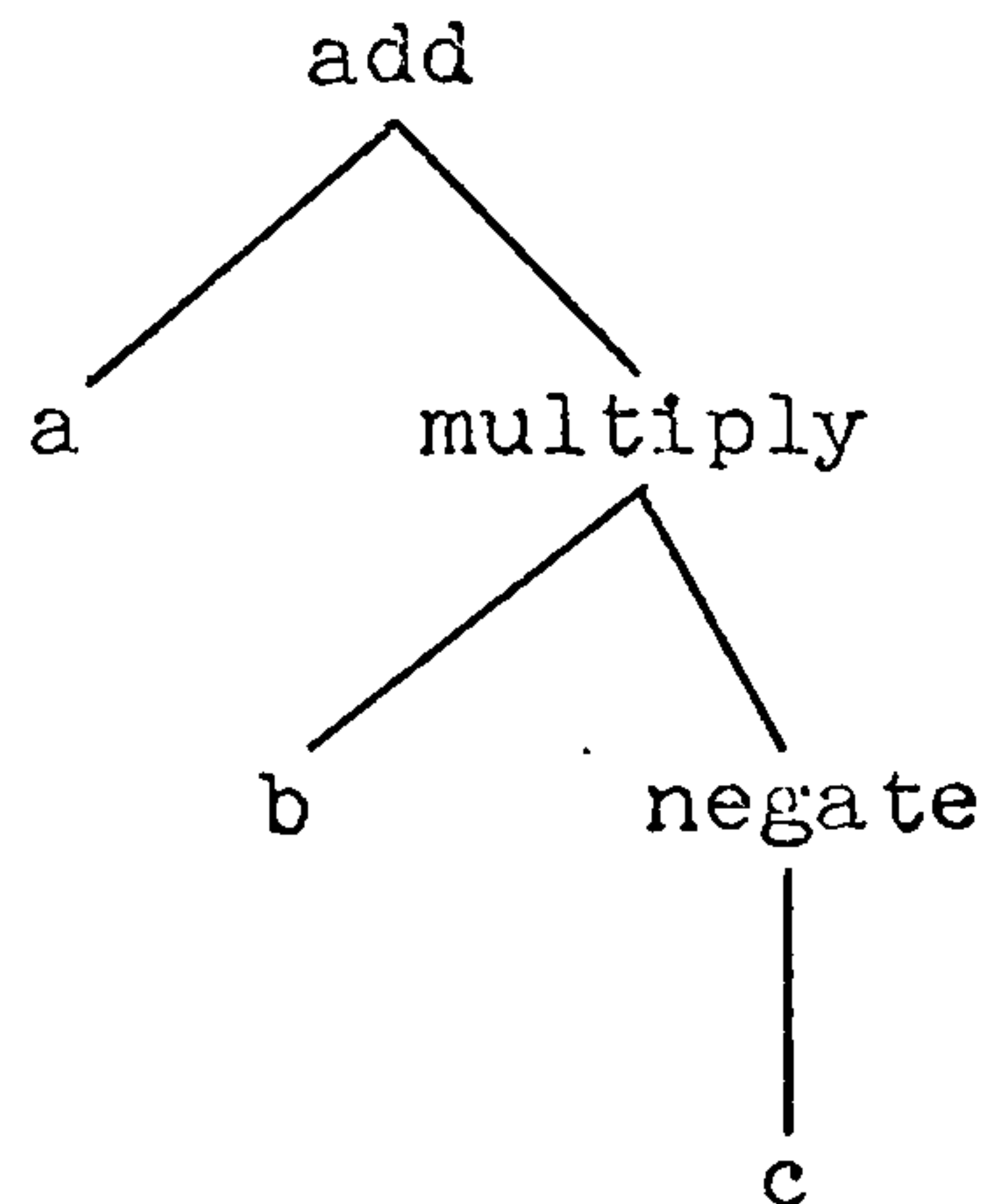


Figure 5.2 Translation of the expression
a + b * (- c)
into (a) postfix form, (b) tree form,
by the EAG of figure 5.1. (c) shows the
operator tree of this expression.

facilitate a rigorous specification of the relationship between the formal syntax and formal semantics of a programming language.

5.2. Notation

We have not stressed our choice of notation for EAGs. We have used a particular notation whose chief merit is to make explicit the essential structure of each hypernotation and protonotion by separating the affix-positions from one another and from the head.

There is at least a possibility that this visual emphasis on structure is unnecessary or undesirable in a language definition intended for programmers (rather than implementors). In order to illustrate the possibilities of non-trivial notational variation, we describe here an alternative notation for EAGs which in style is very similar to van Wijngaarden form (VWF).

Given an EAG, let h be a new, distinguished, affix-nonterminal. For each nonterminal symbol v , there will be exactly one new affix-rule; and if the domains of the affix-positions of v are specified by $\alpha_v = (a_1, \dots, a_N)$ then this affix-rule will be of the form

$$h : t_0 a_1 t_1 \dots a_N t_N \quad ,$$

where t_0, t_1, \dots, t_N are strings of affix-terminals. We require that the CFG $G_h = (A_t, A_n, h, R)$ be unambiguous. We also require now that the sets of terminals and of affix-terminals be disjoint. A hypernotation of the EAG will now simply be a string derivable from h , with each affix-nonterminal replaced by a suitable affix-variable. (And every protonotion will be a terminal production of h .) The hypernotation written as $v(f_1, \dots, f_N)$ in our usual notation will now be written as $t_0 f_1 t_1 \dots f_N t_N$. Adjacent hypernotations and terminals in the right side of a meta-rule will be separated by commas.

Additional affix-rules:-

h : block ;
declarations LIST ;
statements in context LIST ;
statement in context LIST ;
variable in context LIST with type TYPE ;
tag TAG ;
type TYPE ;
in context LIST identifier TAG is TYPEQ ;
TAG and TAG are different .

Meta-rules:-

- (p1) block : var , declarations LIST , begin ,
statements in context LIST , end .
- (p2) declarations : .
- (p3) declarations LIST TAG TYPE : declarations LIST ,
tag TAG , : , type TYPE , : ,
in context LIST identifier TAG is
undefined .
- (p4) statements in context LIST :
statement in context LIST ;
- (p5) statements in context LIST , : ,
statement in context LIST .
- (p6) statement in context LIST : variable in context
LIST with type TYPE , := , variable
in context LIST with type TYPE .
- (p7) variable in context LIST with type TYPE :
tag TAG , in context LIST identifier TAG
is TYPE ;
- (p8) variable in context LIST with type
array with SIMPLE subscript of TYPE ,
[, variable in context LIST with
type SIMPLE ,] .

(continued)

Figure 5.3 The EAG of figure 4.11 written in VWF-like notation.

- (p9) tag w : w .
- (p10) tag x : x .
- (p11) tag y : y .
- (p12) tag z : z .
- (p13) type integer : integer .
- (p14) type boolean : boolean .
- (p15) type array with SIMPLE subscript of TYPE :
 array , [, type SIMPLE ,] , of ,
 type TYPE .
- (p16) in context LIST TAG TYPE identifier TAG is TYPE :
 .
- (p17) in context LIST TAG1 TYPE1 identifier TAG is TYPEQ
 : TAG and TAG1 are different , in context
 LIST identifier TAG is TYPEQ .
- (p18) in context identifier TAG is undefined : .

Figure 5.3 (concluded)

The terminals w, x, y, z, integer and boolean are underlined here to distinguish them from affix-terminals.

Figure 5.3 shows how the EAG of figure 4.11 might be expressed in notation of this style. We believe that a little care on the part of the writer can produce a grammar which is quite readable and almost informal. The reader need not concern himself with affix-positions or their types or domains; he need only have the list of affix-rules to hand while reading the meta-rules.

An essential feature of our alternative notation is that the structure of each hypernotation can be automatically extracted. The hypernotation is parsed using the affix-rules, and the last affix-rule applied in the parse (namely that with *h* on its left side) determines the nonterminal which is the head of the hypernotation. The affix forms are the substrings derived from the affix-nonterminals occurring in the right side of that affix-rule. Since our EAG-to-AG convertor must parse the affix forms anyway, it could accept hypernotations in either notation equally easily.

The principal difference between EAGs expressed in VWF-like notation and VWF itself is the requirement in the former that each hypernotation can be derived from a unique affix-nonterminal and that this derivation reflect the essential structure of the hypernotation. In VWGs hypernotations have no structure in this sense, and a parser constructed from a VWG would have to be able to re-parse the hypernotations at parse-time. We believe that our restriction, relative to VWGs, is quite natural, and we have demonstrated its effectiveness, and therefore we believe that EAGs may be an acceptable alternative to VWGs.

5.3. Further Research

The most obvious lines of further research, perhaps, are those suggested by the results of section 4.7. Although it is very unlikely that a parser constructed from an EAG could closely approach one based on a CFG in terms of compactness and speed, we believe that further refinements to the former could make its performance quite respectable.

For a start, stack overheads could be reduced by merging of code sequences generated from consecutive copy/predicate transitions. Such merging would be along the lines of the first optimisation of section 4.6, in which consecutive copy- and predicate-transitions were merged. In addition, the AG meta-rules produced by our EAG-to-AG convertor at the first stage of parser construction could be re-ordered (subject to condition c3 in the definition of a well-formed AG) in such a way as to make as many primitive predicate hypernotations as possible occur consecutively in each AG meta-rule, and thus increase the applicability of this code optimisation.

A possibility which we have not considered in this thesis is the use of global affixes. Knuth (Knuth 71a) used what were in effect global affixes, as well as inherited and derived affixes, to describe semantics. Our more formal definition of EAGs does not seem to permit any natural extension to handle global affixes explicitly. In an AG the functions can certainly be programmed to access global variables, although the possible side-effects might inhibit certain parser-construction and parsing strategies, such as re-ordering of hypernotations in a meta-rule. To return to EAGs, it is interesting to speculate whether it may be possible to detect automatically what affixes can usefully be held in global locations rather than on the affix stack. One example of such an affix was encountered in section 4.7.

On the subject of efficiency, a theoretical investigation into the computational complexity of parsers constructed from AGs and EAGs might yield results of interest. We strongly suspect that the results would compare favourably with those for other classes of grammars of comparable power.

We defined A-LR(k) grammars in chapter 2, but did not attempt to obtain any results for general A-LR(k) grammars. We extended our AF-LR(k) parsing method a little by allowing multi-predicate states under defined conditions. This, although in practice a useful enhancement, we feel only scratched the surface of a whole new area worthy of investigation. Quite possibly, a further classification of auxiliary grammars, intermediate between AF-LR(k) and A-LR(k),

limiting the number of affixes which need be examined before making a parsing decision, would be useful.

Finally, in investigating the parsing problem, we have concentrated exclusively on well-formed AGs and EAGs. We have already mentioned, in section 5.1, the possibility that non-well-formed EAGs have a part to play in language definition. It would therefore be interesting to know whether our parsing method can be extended to handle non-well-formed AGs or EAGs. We are thinking in terms of AGs which do not satisfy condition c3 (section 1.2) and EAGs which do not satisfy condition (b) (section 3.2). We should, of course, expect a degradation in performance, but this might be acceptable if the grammar has only one or two non-well-formed meta-rules.

APPENDIX A

AN EXTENDED AFFIX GRAMMAR FOR PASCAL

In this appendix we present a definition by an extended affix grammar (EAG) of the programming language PASCAL (Wirth 71, 73). This is intended to serve as a practical example of the use of an EAG for language definition. In addition it can be used to provide a good illustration of how one could build a practical syntax analyser from an EAG.

A.1. The Definition of PASCAL

PASCAL was first described in (Wirth 71). Subsequently the language was improved by various revisions. The version we have chosen to define is the most recent (Wirth 73). The syntax of PASCAL was defined partly by a context-free grammar and partly in English. The disadvantages of this method of definition were well exemplified: although the context-sensitive features of the syntax were in the main described with clarity and care, several points were open to more than one interpretation or were omitted altogether. In each such case we adopted what seemed the most likely interpretation.

There are two principal reasons for our choice of PASCAL for our example. Firstly, the language includes highly systematic data-structuring facilities, which enable us to demonstrate fully the expressive power of EAGs. Secondly, the language was designed to facilitate single-pass compilation by requiring that the

declaration of each identifier textually precede its use. This same feature also facilitates the definition of the language by a fairly clear well-formed EAG. In fact, PASCAL allows some exceptions to this rule; for example, a type identifier may occur in a type declaration preceding its own declaration. In order to avoid complicating our grammar (although we could have done so), we have enforced the strict declaration-before-use requirement. Apart from this, the only restriction we have imposed upon the language is on the allowable syntactic positions of the null pointer and of the empty set, whose types can be determined only by context.

The EAG defining PASCAL is shown in figure A.1. Our usual notational conventions are used :-

- (i) each nonterminal symbol is represented by a sequence of lower-case letters and hyphens;
- (ii) each terminal symbol is represented by a single letter, digit, or special symbol (and some terminals are underlined to distinguish them from EAG metasymbols, namely the parentheses, comma, colon, semicolon and period);
- (iii) each affix-nonterminal is represented by a sequence of upper-case letters;
- (iv) each affix-terminal is represented by a sequence of lower-case letters;
- and (v) each affix-variable is represented by the same sequence of letters as its associated affix-nonterminal, possibly followed by a digit.

The sets of affix-terminals and affix-nonterminals are not given explicitly, but may be deduced from the affix-rules of figure A.1(a). The role of the upper-level grammar in this particular EAG is explained by commentary (enclosed between braces) to the affix-rules.

A list of the terminals of PASCAL is given in figure A.1(b).

The control of the EAG, shown in figure A.1(c), includes two

primitive predicates, each of which checks its two inherited affixes for inequality; they differ only in the domains of their affix-positions. These symbols could have been made nonterminals and their functions defined by meta-rules, but this would have been excessively tedious.

The meta-rules of the EAG are given in figure A.1(d). Again, explanatory commentary (enclosed between braces) is interspersed among the meta-rules. This commentary is aimed at a reader already fairly familiar with the language, and is intended to show how the context-sensitive constraints described in English in (Wirth 73) are defined rigorously by the EAG.

A.2. Syntax Analysis of PASCAL based on the EAG

In a practical translator it is desirable for as much work as possible to be done by a lexical analyser based on finite-state techniques. This tends to optimise the compactness and speed of the syntax analyser and of the translator as a whole. In the EAG of PASCAL, the nonterminals 'tag', 'unsigned-integer', 'unsigned-real', 'character' and 'string' should be re-designated as terminals. Meta-rules p63.1 - p65.10 and p73.1 - p83.2 may then be removed, although they serve to define the actions of the lexical analyser in dealing with these constructs. Each of these symbols, except 'unsigned-real', has one derived affix, and the lexical analyser must stack the representation of such a symbol's affix when that symbol is read by the parser.

Several nonterminals are prime candidates for re-designation as primitive predicate symbols, namely: (i) those concerned with numbers - 'reverse-sign', 'less'; (ii) those concerned with identification - 'identify', 'local', 'identify-label', 'identify-local-label', 'check-labels'. As a result, meta-rules p29.1 - p30.2, p72.1 - p72.3, p61.1 - p62.3, and p67.1 - p69.3 may be removed, although they serve to define the associated functions of

the primitive predicates.

Affixes in the domain $L(\text{TAG})$ are synthesised during the scanning of an identifier tag (meta-rules p63.1 -p63.3), and are never analysed, only compared. Thus, since we have re-designated 'tag' as a terminal, each such affix can be represented by a unique integer (unique to a particular program, that is), the mapping being performed by a hashing function in the lexical analyser.

Each affix ' l^n ' in the domain $L(N)$ should be represented by the non-negative integer n . Similarly, each affix in the domain $L(\text{VALUE})$ should be represented by an integer: ' l^n ' by n , 'minus l^n ' by $-n$.

Most nonterminals in the EAG have a first affix whose domain is $L(C)$. The structure of the language suggests that such affixes should be held in global locations rather than in the parser's affix stack. The same applies to affixes in $L(LC)$ and to some affixes in $L(\text{LIST})$ and $L(\text{LABELS})$.

```

(r1.1-r1.6)  TYPE      : SIMPLE ; pointer to TYPE ; PACKED array with SIMPLE subscript of
                  TYPE ; PACKED record with fields LIST ; PACKED set of SIMPLE ;
                  PACKED file of TYPE .

(r2.1)        SIMPLE   : RANGE SCALAR .

(r3.1-r3.4)  SCALAR    : ARITH ; boolean ; character ; scalar with denotations TAGLIST .

(r4.1-r4.2)  ARITH     : integer ; real .

(r5.1-r5.2)  RANGE     : ; subrange from VALUE to VALUE of .

(r6.1-r6.2)  PACKED    : ; packed .

(r7.1-r7.2)  TAGLIST   : TAG ; TAGLIST and TAG .

{The set of terminal productions of 'TYPE' is the set of PASCAL data types.}

(r8.1-r8.8)  MODE      : CONST ; TYPE variable ; TYPE FUNCTION with LIST parameters ;
                        procedure with LIST parameters ; TYPE FORMAL ;
                        formal procedure ; TYPE field ; TYPE declarer .

(r9.1-r9.2)  CONST     : TYPE constant ; SCALAR constant valued VALUE .

(r10.1-r10.2) FUNCTION : function ; local function .

(r11.1-r11.2) FORMAL   : formal value ; formal variable ; formal procedure .

```

(continued)

Figure A.1(a) Affix-rules of the EAG defining PASCAL.

{Each terminal production of 'MODE' is a mode, which summarises all the syntactic attributes of an identifier, e.g. its type (if any), whether it is a constant, variable, function, etc.}

(r12.1-r12.2)	C	:		; C / LIST .
(r13.1-r13.2)	LIST	:		; LIST TAG MODE .
(r14.1-r14.2)	MODEQ	:	MODE	; undefined .

{Each terminal production of 'LIST' is a list of tag-mode pairs. Such a list may represent

- (i) the list of identifiers declared within a single scope of the program;
- (ii) the list of field identifiers of a record type (r1.4) (in which case every mode will be a terminal production of 'TYPE field');
- or (iii) the list of formal parameters of a function or procedure (r8.3, r8.4) (in which case every mode is either 'formal procedure' or a terminal production of 'TYPE FORMAL').}

{Each terminal production of 'C' is a sequence of lists of tag-mode pairs, the lists being separated by obliques. Each such sequence represents the lists of all identifiers declared in the local scope and in enclosing scopes (as seen from a particular point of the program), the local scope list being the rightmost in the sequence.}

(r15.1-r15.2)	LC	:		; LC / LABELS .
(r16.1-r16.2)	LABELS	:		; LABELS label VALUE .

Figure A.1(a) (continued)

(r17.1-r17.2) LABELQ : label ; undefined .

{Each terminal production of 'LABELS' is a list of labels (either statement labels or case-labels), and each terminal production of 'LC' is a sequence of such lists separated by obliques. 'LABELS' and 'LC' are roughly analogous to 'LIST' and 'C' respectively.}

(r18.1-r18.2) VALUE : N ; minus 1 N .

(r19.1-r19.2) N : ; 1 N .

(r20.1-r20.2) TAG : ALPHA ; TAG ALPHA .

(r21.1-r21.36) ALPHA : a ; b ; ; z ; 0 ; 1 ; ... ; 9 .

Figure A.1(a) (concluded)

{Letters}

a b c d e f g h i j k l m n o p q r s t u v w x y z

{Digits}

0 1 2 3 4 5 6 7 8 9

{Denotation symbols}

. " nil

{Declaration symbols}

label const var type array record set file function procedure packed of

{Operators}

+ - * / div mod v \wedge \neg > \geq = \neq \leq < in \uparrow

{Syntactic symbols}

() [] := , ; : if then else case repeat until while do for to downto begin end with goto

Figure A.1(b) Terminal symbols of PASCAL.

Nonterminal	Domains of inherited affix-positions	Domains of derived affix-positions	Group
actual-parameter	C, MODE		p54
actual-parameters	C, LIST		p52
actual-parameters-list	C, LIST		p53
add	N, N	N	p82
array-type-tail	C, LIST	LIST, TYPE	p22
assignment-compatible	TYPE, TYPE		p51
block	C, LC, LIST		p2
case-label-list	C, SIMPLE, LABELS	LABELS	p35
case-list	C, SIMPLE, LC, LABELS	LABELS, LABELS	p34
character		VALUE	p73
check-labels	LABELS, LABELS		p67
constant	C	CONST	p70
constant-definition-part	C, LIST	LIST	p5
constant-definition-sequence	C, LIST	LIST	p6
declare-scalar-constants	LIST, TAGLIST, SCALAR	N, LIST	p21
digit		N	p81
digit-alpha		ALPHA	p65
div-operator			p43
element-list	C	LIST	p42
equality-operator			p47
expression	C	TYPE	p38

Figure A.1(c) Control of the EAG defining the syntax of PASCAL.

On the right of each line is the number of the group of meta-rules on whose left side the nonterminal occurs.

..

Nonterminal	Domains of inherited affix-positions	Domains of derived affix-positions	Group
factor	C	TYPE	p41
field-declaration	C, LIST, LIST	LIST, LIST, TYPE	p25
field-list	C, LIST, LIST	LIST, LIST	p23
fixed-part	C, LIST, LIST	LIST, LIST	p24
formal-actual-parameter	C		p57
formal-actual-parameters	C		p55
formal-actual-parameters-list	C		p56
formal-declaration	C, FORMAL, LIST	LIST, TYPE	p16
formal-parameters-list	C	LIST	p14
formal-parameters-part	C	LIST	p13
formal-parameters-section	C, LIST	LIST	p15
formal-procedure-declaration	LIST	LIST	p17
fraction			p79
identifier	C	MODE	p60
identify	C, TAG	MODE	p61
identify-label	C, VALUE		p68
identify-local-label	LABELS, VALUE	LABELQ	p69
label	LC		p66
label-declaration		LABELS	p4
label-declaration-part		LABELS	p3
less	VALUE, VALUE		p29
lessp	N, N		p30
letter		ALPHA	p64
local	LIST, TAG	MODEQ	p62

Figure A.1(c) (continued)

Nonterminal	Domains of inherited affix-positions	Domains of derived affix-positions	Group
multiply	N,N	N	p83
ordering-operator			p48
packed-option program	{** distinguished **}	PACKED	p28 p1
record-variable-list	C	C	p36
relational-operator			p46
result-type	C	TYPE	p18
reverse-sign	VALUE	VALUE	p72
routine-declaration-part	C,LC,LIST	LIST	p12
scale-factor			p80
set-relational-operator			p49
sign			p44
simple-expression	C	TYPE	p39
slice	C	TYPE	p59
source	C,TYPE		p50
statement	C,LC,LABELS	LABELS	p31
statement-sequence	C,LC,LABELS	LABELS	p33
step			p37
string		N	p74
string-item		VALUE	p76
string-item-sequence		N	p75

Figure A.1(c) (continued)

Nonterminal	Domains of inherited affix-positions	Domains of derived affix-positions	Group
tag		TAG	p63
tag-list		TAGLIST	p20
term		TYPE	p40
type	C	LIST, TYPE	p19
type-definition-part	C, LIST	LIST	p7
type-definition-sequence	C, LIST	LIST	p8
union-operator			
unlabelled-statement	C, LC, LABELS	LABELS	p45
unsigned-constant	C	CONST	p32
unsigned-integer		N	p71
unsigned-real			p77
			p78
variable	C	TYPE	p58
variable-declaration	C, LIST	LIST, TYPE	p11
variable-declaration-part	C, LIST	LIST	p9
variable-declaration-sequence	C, LIST	LIST	p10
variant	C, LIST, LIST, SIMPLE, LABELS	LIST, LIST, LABELS	p27
variant-part	C, LIST, LIST	LIST, LIST, SIMPLE, LABELS	p26
Primitive predicate	Domains of inherited affix-positions	Domains of derived affix-positions	Associated function
unequal-tag	TAG, TAG		$\lambda t \lambda u (t \neq u)$
unequal-value	VALUE, VALUE		$\lambda v \lambda w (v \neq w)$

Figure A.1(c) (concluded)

{THE PROGRAM - continued}

{The program is considered as a procedure to which the identifiers input and output are formal parameters. The identifiers false, true, eol, etc., are considered as declared externally to the program. In meta-rule 1.1, EOL and ALFALENG stand for the terminal productions of 'VALUE' which represent the (implementation-dependent) values of the constants eol and alfa leng respectively.}

{BLOCKS}

(p2.1) block(C, LC, LIST) : label-declaration-part(LABELS1)
 constant-definition-part(C, LIST, LIST1)
 type-definition-part(C, LIST1, LIST2)
 variable-declaration-part(C, LIST2, LIST3)
 routine-declaration-part(C, LC / LABELS1, LIST3, LIST4)
 begin statement-sequence(C / LIST4, LC / LABELS1, , LABELS2) end
 check-labels(LABELS1, LABELS2) .

{The first affix of 'block' is the external context of the block, i.e. the list of all identifiers declared in declarations and formal-parameters-lists external to the program or routine-declaration containing the block. The second affix is the list of labels declared externally to the block. The third affix is the list of identifiers declared in any formal-parameters-list local to the block.}

Figure A.1(d) (continued)

{BLOCKS - continued}

{The 'check-labels' hypernotation in meta-rule p2.1 ensures that every label declared in the local label-declaration-part occurs as a statement label in the statement-sequence.}

{DECLARATIONS}

(p3.1) label-declaration-part() : .
(p3.2) label-declaration-part(LABELS) : label label-declaration(LABELS) .
(p4.1) label-declaration(label N) : unsigned-integer(N) .
(p4.2) label-declaration(LABELS label N) : label-declaration(LABELS) , unsigned-integer(N) .
(p5.1) constant-definition-part(C, LIST, LIST) : .
(p5.2) constant-definition-part(C, LIST1, LIST2) : const constant-definition-sequence(C, LIST1, LIST2) .

Figure A.1(d) (continued)

{DECLARATIONS - continued}

- (p6.1) constant-definition-sequence(C, LIST, LIST TAG CONST) : tag(TAG)
 local(LIST, TAG, undefined) = constant(C / LIST, CONST) ; .
- (p6.2) constant-definition-sequence(C, LIST1, LIST2 TAG CONST) :
 constant-definition-sequence(C, LIST1, LIST2) tag(TAG)
 local(LIST2, TAG, undefined) = constant(C / LIST2, CONST) ; .
- (p7.1) type-definition-part(C, LIST, LIST) : .
- (p7.2) type-definition-part(C, LIST1, LIST2) : type type-definition-sequence(C, LIST1, LIST2) .
- (p8.1) type-definition-sequence(C, LIST1, LIST2 TAG TYPE declarer) :
 tag(TAG) = type(C, LIST1, LIST2, TYPE) ;
 local(LIST2, TAG, undefined) ;
- (p8.2) type-definition-sequence(C, LIST1, LIST) tag(TAG) =
 type(C, LIST, LIST2, TYPE) local(LIST2, TAG, undefined) ; .

{DECLARATIONS - continued}

```
(p9.1)  variable-declaration-part(C, LIST, LIST)      :      .
(p9.2)  variable-declaration-part(C, LIST1, LIST2)    :  var variable-declaration-sequence(C,
                                                         LIST1, LIST2)  .

(p10.1) variable-declaration-sequence(C, LIST1, LIST2) :
        variable-declaration(C, LIST1, LIST2, TYPE) ; ;
        variable-declaration-sequence(C, LIST1, LIST)
        variable-declaration(C, LIST, LIST2, TYPE) ; .

(p10.2)

(p11.1) variable-declaration(C, LIST1, LIST2 TAG TYPE variable, TYPE) :
        tag(TAG) : type(C, LIST1, LIST2, TYPE)
        local(LIST2, TAG, undefined) ;

(p11.2)
        tag(TAG) , variable-declaration(C, LIST1, LIST2, TYPE)
        local(LIST2, TAG, undefined) .
```

Figure A.1(d) (continued)

{DECLARATIONS - continued}

{The first affix of each of 'constant-definition-part', 'constant-definition-sequence', 'type-definition-part', 'type-definition-sequence', 'variable-declaration-part', 'variable-declaration-sequence' and 'variable-declaration' is the external context of the block containing these constructs. The second (resp., third) affix of each is the list of identifiers declared locally, up to but excluding (resp., up to and including) the construct. (The third affix of each is an example of a derived affix which is partly determined by the left context.)}

```
(p12.1)  routine-declaration-part(C, LC, LIST, LIST) :      .

(p12.2)  routine-declaration-part(C, LC, LIST1, LIST2 TAG procedure with LIST parameters) :

        routine-declaration-part(C, LC, LIST1, LIST2)  procedure
            tag(TAG)  local(LIST2, TAG, undefined)
            formal-parameters-part(C / LIST1, LIST)  ;
            block(C / LIST2 TAG procedure with LIST parameters, LC, LIST) ; .

(p12.3)  routine-declaration-part(C, LC, LIST1, LIST2 TAG TYPE function with LIST parameters) :

        routine-declaration-part(C, LC, LIST1, LIST2)  function
            tag(TAG)  local(LIST2, TAG, undefined)
            formal-parameters-part(C / LIST1, LIST)  :
            result-type(C / LIST1, TYPE) ;
            block(C / LIST2 TAG TYPE local function with LIST parameters,
                    LC, LIST) ; .
```

Figure A.1(d) (continued)

{DECLARATIONS - continued}

{The first, third and fourth affixes of 'routine-declaration-part' are similar to the first, second and third affixes (respectively) of 'constant-definition-part', etc. The second affix of 'routine-declaration-part' is the list of all labels with currently valid declarations, including any declared in a local label-declaration-part.}

{In meta-rule p12.3, the function identifier is added to the external context of the nested block (which is the body of the function) as a "local function" but is added to the list of identifiers declared in the enveloping block as an ordinary function. This distinction, in conjunction with p32.2, ensures that only within the body of the function may an assignment be made to the function identifier.}

(p13.1) formal-parameters-part(C,) : .
(p13.2) formal-parameters-part(C, LIST) : [formal-parameters-list(C, LIST)] .
(p14.1) formal-parameters-list(C, LIST) : formal-parameters-section(C, , LIST) ;
(p14.2) : formal-parameters-list(C, LIST1) ;
 formal-parameters-section(C, LIST1, LIST) .

{The first affix of each of 'formal-parameters-part' and 'formal-parameters-list' is the list of all identifiers whose declarations are valid at the start of the enveloping routine-declaration-part. The second affix of each is the list of formal parameters declared in it.}

{DECLARATIONS - continued}

```

(p15.1)  formal-parameters-section(C, LIST1, LIST2) :
          formal-declaration(C, formal value, LIST1, LIST2, TYPE) ;

(p15.2)  var formal-declaration(C, formal variable, LIST1, LIST2, TYPE) ;

(p15.3)  function formal-declaration(C, formal function, LIST1, LIST2, TYPE) ;

(p15.4)  procedure formal-procedure-declaration(LIST1, LIST2) .

(p16.1)  formal-declaration(C, FORMAL, LIST, LIST TAG TYPE FORMAL, TYPE) : tag(TAG)
          local(LIST, TAG, undefined) : identifier(C, TYPE declarer) .

(p16.2)  formal-declaration(C, FORMAL, LIST1, LIST2 TAG TYPE FORMAL, TYPE) : tag(TAG) 2
          formal-declaration(C, FORMAL, LIST1, LIST2, TYPE)
          local(LIST2, TAG, undefined) .

(p17.1)  formal-procedure-declaration(LIST, LIST TAG formal procedure) : tag(TAG)
          local(LIST, TAG, undefined) .

(p17.2)  formal-procedure-declaration(LIST1, LIST2 TAG formal procedure) :
          formal-procedure-declaration(LIST1, LIST2) 2 tag(TAG)
          local(LIST2, TAG, undefined) .

```

Figure A.1(d) (continued)

{DECLARATIONS - continued}

{The hypernotations of the form 'local(LIST, TAG, undefined)' in meta-rules p6.1, p6.2, p8.1, p8.2, p11.1, p11.2, p12.2, p12.3, p16.1, p16.2, p17.1 and p17.2, and also in p21.1 and p21.2, ensure that no identifier is declared more than once in the same block (including any local formal-parameters-list).}

(p18.1) result-type(C, SIMPLE) : identifier(C, SIMPLE declarer) .

(p18.2) result-type(C, pointer to TYPE) : identifier(C, pointer to TYPE declarer) .

{Meta-rules p18.1 and p18.2 ensure that the type of a function is either a simple type or a pointer type.}

{TYPES}

(p19.1) type(C, LIST, TYPE) : identifier(C / LIST, TYPE declarer) .

(p19.2) type(C, LIST, LIST, subrange from VALUE1 to VALUE2 of SCALAR) :
 constant(C / LIST, SCALAR constant valued VALUE1)
 constant(C / LIST, SCALAR constant valued VALUE2) less(VALUE1, VALUE2) .

Figure A.1(d) (continued)

{TYPES - continued}

```
(p19.3)  type(C, LIST1, LIST2, scalar with denotations TAGLIST) : ( tag-list(TAGLIST) )
          declare-scalar-constants(LIST1, TAGLIST, scalar with denotations TAGLIST, N, LIST2) .

(p19.4)  type(C, LIST1, LIST2, pointer to TYPE) : ↑ type(C, LIST1, LIST2, TYPE) .

(p19.5)  type(C, LIST1, LIST2, PACKED array with SIMPLE subscript of TYPE) : packed-option(PACKED)
          array [ type(C, LIST1, LIST, SIMPLE)
          array-type-tail(C, LIST, LIST2, TYPE) .

(p19.6)  type(C, LIST1, LIST2, PACKED record with fields LIST) : packed-option(PACKED)
          record field-list(C, LIST1, , LIST2, LIST) end .

(p19.7)  type(C, LIST1, LIST2, PACKED set of SIMPLE) : packed-option(PACKED) set of
          type(C, LIST1, LIST2, SIMPLE) .

(p19.8)  type(C, LIST1, LIST2, PACKED file of TYPE) : packed-option(PACKED) file of
          type(C, LIST1, LIST2, TYPE) .
```

{The first affix of 'type' is the external context of the block in which the type occurs. The second (resp., third) affix is the list of identifiers declared locally, up to but excluding (resp., up to and including) the type. (These lists will be different if the type is, or has a component type which is, a previously undeclared scalar type: the presence of the latter constitutes implicit declarations of the constants of that scalar type.) The fourth affix of 'type' is the represented type.}

Figure A.1(d) (continued)

{TYPES - continued}

```

(p20.1) tag-list(TAG) : tag(TAG) .

(p20.2) tag-list(TAGLIST and TAG) : tag-list(TAGLIST) , tag(TAG) .

(p21.1) declare-scalar-constants(LIST, TAG, SCALAR, , LIST TAG SCALAR constant valued ) :
        local(LIST, TAG, undefined) .

(p21.2) declare-scalar-constants(LIST1, TAGLIST and TAG, SCALAR, 1 N, LIST2 TAG SCALAR constant
        valued 1 N) :
        declare-scalar-constants(LIST1, TAGLIST, SCALAR, N, LIST2)
        local(LIST2, TAG, undefined) .

{Meta-rules p19.3, p21.1 and p21.2 handle new scalar types. The identifiers
 occurring in the tag-list are added to the list of locally declared identifiers as
 constants of the new scalar type, with their values represented by '', '1', '1 1',
 etc.}

(p22.1) array-type-tail(C, LIST1, LIST2, TYPE) : 1 of type(C, LIST1, LIST2, TYPE) .

(p22.2) array-type-tail(C, LIST1, LIST2, array with SIMPLE subscript of TYPE) :
        type(C, LIST1, LIST, SIMPLE) array-type-tail(C, LIST, LIST2, TYPE) .

```

Figure A.1(d) (continued)

{TYPES - continued}

```
(p23.1)  field-list(C, LIST1, LIST7, LIST2, LIST3) : fixed-part(C, LIST1, LIST7, LIST2, LIST8) ;
(p23.2)  fixed-part(C, LIST1, LIST7, LIST3, LIST9) ;
          variant-part(C, LIST3, LIST9, LIST2, LIST8, SIMPLE, LABELS) ;
(p23.3)  variant-part(C, LIST1, LIST7, LIST2, LIST8, SIMPLE, LABELS) .

(p24.1)  fixed-part(C, LIST1, LIST7, LIST8) :
          field-declaration(C, LIST1, LIST7, LIST2, LIST8, TYPE) ;
(p24.2)  fixed-part(C, LIST1, LIST7, LIST3, LIST9) ;
          field-declaration(C, LIST3, LIST9, LIST2, LIST8, TYPE) .

(p25.1)  field-declaration(C, LIST1, LIST2, LIST TAG TYPE field, TYPE) : tag(TAG)
          local(LIST, TAG, undefined) : type(C, LIST1, LIST2, TYPE) .
(p25.2)  field-declaration(C, LIST1, LIST7, LIST2, LIST8 TAG TYPE field, TYPE) : tag(TAG)
          field-declaration(C, LIST1, LIST7, LIST2, LIST8, TYPE)
          local(LIST8, TAG, undefined) .
```

Figure A.1(d) (continued)

{TYPES - continued}

(p26.1) variant-part(C, LIST1, LIST7, LIST2, LIST8, SIMPLE, LABELS) :

case tag(TAG) local(LIST7, TAG, undefined) :
 identifier(C / LIST1, SIMPLE declarer) of
 variant(C, LIST1, LIST7 TAG SIMPLE field, SIMPLE, ,
 LIST2, LIST8, LABELS) ;

(p26.2)

 variant-part(C, LIST1, LIST7, LIST3, LIST9, SIMPLE, LABELS1) ;
 variant(C, LIST3, LIST9, SIMPLE, LABELS1, LIST2, LIST8, LABELS) .

{The first affix of each of 'field-list', 'fixed-part', 'field-declaration' and 'variant-part' is the external context of the block in which the enveloping record-type occurs. The second (resp., fourth) affix of each is the list of identifiers declared locally, up to but excluding (resp., up to and including) the construct. The third (resp., fifth) affix of each is the list of field identifiers declared in the record-type, up to but excluding (resp., up to and including) the construct. The sixth affix of 'variant-part' is the (simple) type of the tag field. The seventh affix of 'variant-part' is the list of the values of the constants occurring in it as case-labels.}

{The hypernotations of the form 'local(LIST, TAG, undefined)' in p25.1, p25.2 and p26.1 ensure that no two fields of the same record-type have the same identifiers.}

{TYPES - continued}

```
(p27.1) variant(C, LIST1, LIST7, SIMPLE, LABELS1, LIST1, LIST7, LABELS2) :
        case-label-list(C, SIMPLE, LABELS1, LABELS2) .

(p27.2) variant(C, LIST1, LIST7, SIMPLE, LABELS1, LIST2, LIST8, LABELS2) :
        case-label-list(C, SIMPLE, LABELS1, LABELS2) :
        [ field-list(C, LIST1, LIST7, LIST2, LIST8) ] .
```

{The fifth (resp., eighth) affix of 'variant' is the list of the values of the constants occurring as case-labels in the enveloping variant-part, up to but excluding (resp., up to and including) the variant. Meta-rules p35.1 and p35.2 ensure that no value occurs more than once as a case-label in the variant-part.}

```
(p28.1) packed-option(packed) : packed .

(p28.2) packed-option( ) : .

(p29.1) less(N1, N2) : lessp(N1, N2) .

(p29.2) less(minus 1 N1, N2) : .

(p29.3) less(minus 1 N1, minus 1 N2) : lessp(N2, N1) .

(p30.1) lessp( , 1 N) : .

(p30.2) lessp(1 N1, 1 N2) : lessp(N1, N2) .
```

Figure A.1(d) (continued)

{STATEMENTS}

```
(p31.1) statement(C, LC, LABELS1, LABELS2) : unlabelled-statement(C, LC, LABELS1, LABELS2) ;

(p31.2) unsigned-integer(N) identify-local-label(LABELS1, N, undefined)
      : unlabelled-statement(C, LC, LABELS1 label N, LABELS2) .

(p32.1) unlabelled-statement(C, LC, LABELS, LABELS) :
      variable(C, TYPE) := source(C, TYPE) ;

(p32.2) identifier(C, TYPE local function with LIST parameters) :=
      source(C, TYPE) ;

(p32.3) identifier(C, procedure with LIST parameters) actual-parameters(C, LIST) ;

(p32.4) identifier(C, formal procedure) formal-actual-parameters(C) ;

(p32.5) goto label(LC) ;

(p32.6) .

(p32.7) unlabelled-statement(C, LC, LABELS1, LABELS2) :
      begin statement-sequence(C, LC, LABELS1, LABELS2) end ;

(p32.8) if expression(C, boolean) then statement(C, LC, LABELS1, LABELS2) .
```

Figure A.1(d) (continued)

{STATEMENTS - continued}

```
(p32.9)  unlabelled-statement(C, LC, LABELS1, LABELS2)  :  
  
        if expression(C, boolean) then statement(C, LC, LABELS1, LABELS)  
        else statement(C, LC, LABELS, LABELS2) ;  
  
(p32.10) case expression(C, SIMPLE) of  
        case-list(C, SIMPLE, LC, LABELS1, LABELS2, LABELS8) end ;  
  
(p32.11) while expression(C, boolean) do statement(C, LC, LABELS1, LABELS2) ;  
  
(p32.12) repeat statement-sequence(C, LC, LABELS1, LABELS2) until  
        expression(C, boolean) ;  
  
(p32.13) for tag(TAG) identify(C, TAG, SIMPLE variable) := source(C, SIMPLE)  
        step source(C, SIMPLE) do  
        statement(C / TAG SIMPLE constant, LC, LABELS1, LABELS2) ;  
  
(p32.14) with record-variable-list(C, C1) do  
        statement(C1, LC, LABELS1, LABELS2) .  
  
(p33.1)  statement-sequence(C, LC, LABELS1, LABELS2) : statement(C, LC, LABELS1, LABELS2) ;  
  
(p33.2)  statement-sequence(C, LC, LABELS1, LABELS) ;  
        statement(C, LC, LABELS, LABELS2) .
```

Figure A.1(d) (continued)

{STATEMENTS - continued}

{The first (resp., second) affix of each of 'statement', 'unlabelled-statement' and 'statement-sequence' is the list of all identifiers (resp., labels) whose declarations are currently valid. The third (resp., fourth) affix of each is the list of labels which occur as statement-labels in the current block, up to but excluding (resp., up to and including) the construct.}

{The 'identify-local-label' hypernotation in p31.2 ensures that no label occurs more than once as a statement label in the same block.}

{Meta-rules p32.8 and p32.9 together give rise to an ambiguity, which merely reflects an ambiguity in the context-free syntax (Wirth 73, section 9.2.2.1).}

{In meta-rule p32.13, the control variable identifier is made to appear to be a constant within the controlled statement. This ensures that no assignment can be made to it within the latter.}

(p34.1) case-list(C, SIMPLE, LC, LABELS1, LABELS2, LABELS8) :

case-label-list(C, SIMPLE, LABELS8) ;
statement(C, LC, LABELS1, LABELS2) ;

(p34.2)

case-list(C, SIMPLE, LC, LABELS1, LABELS, LABELS7) ;
case-label-list(C, SIMPLE, LABELS7, LABELS8) ;
statement(C, LC, LABELS, LABELS2) .

{STATEMENTS - continued}

```
(p35.1) case-label-list(C, RANGE SCALAR, LABELS, LABELS label VALUE) :  
        unsigned-constant(C, SCALAR constant valued VALUE)  
        identify-local-label(LABELS, VALUE, undefined) .  
  
(p35.2) case-label-list(C, RANGE SCALAR, LABELS7, LABELS8) :  
        unsigned-constant(C, SCALAR constant valued VALUE)  
        identify-local-label(LABELS7, VALUE, undefined)  
        case-label-list(C, RANGE SCALAR, LABELS7 label VALUE, LABELS8) .
```

{The second affix of each of 'case-list' and 'case-label-list' is the (simple) type of the selector of the case-statement. The sixth affix of 'case-list' is the list of the values of the constants which occur in it as case-labels. The third (resp., fourth) affix of 'case-label-list' is the list of the values of the constants which occur as case-labels in the case-list up to but excluding (resp., up to and including) the case-label-list.}

{The 'identify-local-label' hypernotations in meta-rules p35.1 and p35.2 ensure that no value occurs more than once as a case-label in any case-list.}

{STATEMENTS - continued}

(p36.1) record-variable-list(C, C / LIST) : variable(C, record with fields LIST) .
(p36.2) record-variable-list(C1, C2 / LIST) : record-variable-list(C1, C2) ,
variable(C2, record with fields LIST) .

{The record-variable-list in a with-statement alters the context of its component statement by opening the scopes of the field identifiers of the constituent record variables. In conjunction with p58.4, this allows field identifiers to occur as variables.}

(p37.1) step : to ;
(p37.2) downto .

Figure A.1(d) (continued)

•

Figure A.1(d) (continued)

{EXPRESSIONS - continued}

```

(p39.1) simple-expression(C, TYPE) : term(C, TYPE) .
(p39.2) simple-expression(C, ARITH) : sign term(C, RANGE ARITH) ;
(p39.3) simple-expression(C, RANGE1 ARITH) sign term(C, RANGE2 integer) .
(p39.4) simple-expression(C, real) : simple-expression(C, RANGE1 ARITH) sign
      term(C, RANGE2 real) .
(p39.5) simple-expression(C, boolean) : simple-expression(C, boolean) V term(C, boolean) .
(p39.6) simple-expression(C, PACKED set of SCALAR) : simple-expression(C, PACKED set of
      RANGE1 SCALAR) union-operator term(C, PACKED set of RANGE2 SCALAR) .
(p40.1) term(C, TYPE) : factor(C, TYPE) .
(p40.2) term(C, ARITH) : term(C, RANGE1 ARITH) * factor(C, RANGE2 integer) .
(p40.3) term(C, real) : term(C, RANGE1 ARITH) * factor(C, RANGE2 real) ;
(p40.4) term(C, RANGE1 ARITH1) / factor(C, RANGE2 ARITH2) .
(p40.5) term(C, integer) : term(C, RANGE1 integer) div-operator factor(C, RANGE2 integer) .
(p40.6) term(C, boolean) : term(C, boolean) ^ factor(C, boolean) .
(p40.7) term(C, PACKED set of SCALAR) : term(C, PACKED set of RANGE1 SCALAR) ^
      factor(C, PACKED set of RANGE2 SCALAR) .

```

Figure A.1(d) (continued)

{EXPRESSIONS - continued}

```
(p41.1) factor(C, SCALAR) : unsigned-constant(C, SCALAR constant valued VALUE) .
(p41.2) factor(C, TYPE) : unsigned-constant(C, TYPE constant) ;
(p41.3) : variable(C, TYPE) ;
(p41.4) identifier(C, TYPE FUNCTION with LIST parameters)
        actual-parameters(C, LIST) ;
(p41.5) identifier(C, TYPE formal function) formal-actual-parameters(C) ;
(p41.6) [ expression(C, TYPE) ] .
(p41.7) factor(C, boolean) : ~ factor(C, boolean) .
(p41.8) factor(C, set of SIMPLE) : [ element-list(C, SIMPLE) ] .
```

{The second affix of each of 'expression', 'simple-expression', 'term' and 'factor' is its type.}

{Meta-rules p38.2, p39.3, p39.4, p40.2, p40.3, p40.4, p40.5 allow scalar operations to be applied to operands whose types are (possibly different) subranges of the allowable scalar types. Likewise, p38.7, p38.8, p39.6 and p40.7 allow set operations to be applied to operands of set type whose component types differ in range.}

Figure A.1(d) (continued)

{EXPRESSIONS - continued}

(p42.1)	element-list(C, SIMPLE)	:	expression(C, SIMPLE)	.
(p42.2)	element-list(C, SCALAR)	:	element-list(C, RANGE1 SCALAR), expression(C, RANGE2 SCALAR)	.
(p43.1-p43.2)	div-operator	:	$\frac{\text{div}}{\text{mod}}$.
(p44.1-p44.2)	sign	:	+ ; -	.
(p45.1-p45.2)	union-operator	:	V ; -	.
(p46.1-p46.6)	relational-operator	:	> ; \geq ; = ; \neq ; \leq ; <	.
(p47.1-p47.2)	equality-operator	:	= ; \neq	.
(p48.1-p48.4)	ordering-operator	:	> ; \geq ; \leq ; <	.
(p49.1-p49.4)	set-relational-operator	:	= ; \neq ; \geq ; \leq	.

Figure A.1(d) (continued)

{EXPRESSIONS - continued}

- (p50.1) source(C, TYPE) : expression(C, TYPE1) assignment-compatible(TYPE1, TYPE) .
- (p50.2) source(C, pointer to TYPE) : nil .
- (p50.3) source(C, PACKED set of SIMPLE) : [] .
- (p51.1) assignment-compatible(RANGE1 SCALAR, RANGE2 SCALAR) : .
- (p51.2) assignment-compatible(RANGE1 integer, RANGE2 real) : .
- (p51.3) assignment-compatible(pointer to TYPE, pointer to TYPE) : .
- (p51.4) assignment-compatible(PACKED array with SIMPLE subscript of TYPE,
PACKED array with SIMPLE subscript of TYPE) : .
- (p51.5) assignment-compatible(PACKED record with fields LIST, PACKED record with fields LIST) : .
- (p51.6) assignment-compatible(PACKED set of RANGE1 SCALAR, PACKED set of RANGE2 SCALAR) : .

{A "source" is an expression whose type is determined by context. Sources occur in assignment statements (p32.1, p32.2), in for-statements (p32.13), as actual-parameters (p54.1), and as array subscripts (p59.1, p59.2). The types of the empty set (p50.3) and of the null pointer (p50.2) can be determined only when they occur as sources. Allowable type changes in sources are defined by the rules of assignment compatibility (p51.1 - p51.6).}

Figure A.1(d) (continued)

{ACTUAL PARAMETERS}

```

(p52.1)  actual-parameters(C, )      :      .
(p52.2)  actual-parameters(C, LIST) : ( actual-parameters-list(C, LIST) .

(p53.1)  actual-parameters-list(C, TAG MODE)      : actual-parameter(C, MODE) .
(p53.2)  actual-parameters-list(C, LIST TAG MODE) : actual-parameters-list(C, LIST) ,
                                                    actual-parameter(C, MODE) .

(p54.1)  actual-parameter(C, TYPE formal value)   : source(C, TYPE) .
(p54.2)  actual-parameter(C, TYPE formal variable) : variable(C, TYPE) .
(p54.3)  actual-parameter(C, TYPE formal function) : identifier(C, TYPE FUNCTION with LIST
                                                    parameters) ;
(p54.4)                                     identifier(C, TYPE formal function) .
(p54.5)  actual-parameter(C, formal procedure)     : identifier(C, procedure with LIST
                                                    parameters) ;
(p54.6)                                     identifier(C, formal procedure) .

```

{The second affix of each of 'actual-parameters' and 'actual-parameters-list' is the list of identifiers and modes of the formal parameters of the called function or procedure. Only the modes are relevant. The second affix of 'actual-parameter' is

{ACTUAL PARAMETERS - continued}

the mode of the corresponding formal parameter.}

```
(p55.1) formal-actual-parameters(C) : ;  
(p55.2) [ formal-actual-parameters-list(C) ] .  
  
(p56.1) formal-actual-parameters-list(C) : formal-actual-parameter(C) ;  
(p56.2) formal-actual-parameters-list(C) ,  
        formal-actual-parameter(C) .  
  
(p57.1) formal-actual-parameter(C) : expression(C, TYPE) ;  
(p57.2) [ ] ;  
(p57.3) nil ;  
(p57.4) variable(C, TYPE) ;  
(p57.5) identifier(C, TYPE FUNCTION with LIST parameters) ;  
(p57.6) identifier(C, TYPE formal function) ;  
(p57.7) identifier(C, procedure with LIST parameters) :  
(p57.8) identifier(C, formal procedure) .
```


{ACTUAL PARAMETERS - continued}

{Meta-rules p55.1 - p57.8 are invoked when the called function or procedure is itself a formal parameter and therefore the number and modes of its own formal parameters are unknown. Meta-rules p57.1 and p57.4 give rise to an ambiguity, which reflects an ambiguity in the CF syntax (Wirth 73, section 9.1.2).}

{VARIABLES}

```
(p58.1)  variable(C, TYPE) : identifier(C, TYPE variable) ;
(p58.2)                                     identifier(C, TYPE formal value) ;
(p58.3)                                     identifier(C, TYPE formal variable) ;
(p58.4)                                     identifier(C, TYPE field) ;
(p58.5)                                     slice(C, TYPE) ] ;
(p58.6)  variable(C, PACKED record with fields LIST) : tag(TAG)
                                                local(LIST, TAG, TYPE field) ;
(p58.7)                                     variable(C, PACKED file of TYPE) ↑ ;
(p58.8)                                     variable(C, pointer to TYPE) ↑ .
```

Figure A.1(d) (continued)

{VARIABLES - continued}

(p59.1) slice(C, TYPE) : variable(C, array with SIMPLE subscript of TYPE) [source(C, SIMPLE) ;
(p59.2) slice(C, array with SIMPLE subscript of TYPE) , source(C, SIMPLE) .

{The second affix of each of 'variable' and 'slice' is its type.}

{Meta-rule p58.4 can be invoked only within a with-statement (p32.14), since otherwise a field identifier is accessible only in a field designator (p58.6).}

{Meta-rules p58.5, p59.1 and p59.2 allow the number of subscripts to be less than the dimensionality of the array variable; the resulting variable is then still of array type. The same meta-rules prevent subscripting of packed arrays.}

{IDENTIFIERS}

(p60.1) identifier(C, MODE) : tag(TAG) identifier(C, TAG, MODE) .
(p61.1) identify(C / LIST, TAG, MODE) : local(LIST, TAG, MODE) ;
(p61.2) local(LIST, TAG, undefined) identifier(C, TAG, MODE) .

Figure A.1(d) (continued)

{IDENTIFIERS - continued}

```
(p62.1)  local(LIST TAG MODE, TAG, MODE)      :      .
(p62.2)  local(LIST TAG1 MODE, TAG, MODEQ)      :  unequal-tag(TAG1, TAG)  local(LIST, TAG, MODEQ)  .
(p62.3)  local( , TAG, undefined)              :      .

(p63.1)  tag(ALPHA)                          :  letter(ALPHA)  .
(p63.2)  tag(TAG ALPHA)                      :  tag(TAG)  digit-alpha(ALPHA)  ;
(p63.3)                                     tag(TAG)  letter(ALPHA)  .

(p64.1)  letter(a)                          :  a  .
(p64.25) letter(z)                          :  z  .

(p65.1)  digit-alpha(0)                      :  0  .
(p65.10) digit-alpha(9)                      :  9  .
```

Figure A.1(d) (continued)

{IDENTIFIERS - continued}

{The second affix of 'identifier' is its mode. Its mode is what characterises it; it is merely represented in the program by a "tag". The mapping from the tag and its context on to its mode is achieved by meta-rules p61.1 - p62.3.}

{The first affix of 'identify' is the context, a sequence of lists of tag-mode pairs. Each list contains the formal parameters, constant identifiers, type identifiers, variable identifiers, procedure identifiers and function identifiers declared in a currently accessible (i.e. enclosing) scope. The rightmost list in the sequence is that of the local (most deeply nested) scope.}

{Meta-rules p61.1 and p61.2 ensure that the search for the identifier starts with the local scope, and only if that fails does the search continue outwards.}

{The first affix of 'local' is a single list of tag-mode pairs, and the second a tag. If the tag is matched then the third affix is the corresponding mode; otherwise it is 'undefined'.}

{LABELS}

```
(p66.1) label(LC) : unsigned-integer(N) identify-label(LC, N) .

(p67.1) check-labels( , LABELS) : .

(p67.2) check-labels(LABELS1 label VALUE, LABELS2) : check-labels(LABELS1, LABELS2)
        identify-local-label(LABELS2, VALUE, label) .

(p68.1) identify-label(LC / LABELS, VALUE) : identify-local-label(LABELS, VALUE, label) ;

(p68.2) : identify-local-label(LABELS, VALUE, undefined)
        identify-label(LC, VALUE) .

(p69.1) identify-local-label(LABELS label VALUE, label) : .

(p69.2) identify-local-label(LABELS label VALUE1, VALUE, LABELQ) : unequal-value(VALUE1, VALUE)
        identify-local-label(LABELS, VALUE, LABELQ) .

(p69.3) identify-local-label( , VALUE, undefined) : .

        {'identify-label' and 'identify-local-label' handle identification of labels
         but are otherwise similar to 'identify' and 'local' respectively.}
```

Figure A.1(d) (continued)

{CONSTANTS}

- (p70.1) constant(C, CONST) : unsigned-constant(C, CONST) .
- (p70.2) constant(C, integer constant valued VALUE) : + unsigned-constant(C, integer constant valued VALUE) ;
- (p70.3) - unsigned-constant(C, integer constant valued VALUE1) reverse-sign(VALUE1, VALUE) .
- (p70.4) constant(C, real constant) : sign unsigned-constant(C, real constant) .
- (p71.1) unsigned-constant(C, CONST) : identifier(C, CONST) .
- (p71.2) unsigned-constant(C, character constant valued VALUE) : character(VALUE) .
- (p71.3) unsigned-constant(C, integer constant valued N) : unsigned-integer(N) .
- (p71.4) unsigned-constant(C, real constant) : unsigned-real .
- (p71.5) unsigned-constant(C, packed array with subrange from 1 to N of integer subscript of character constant) : string(N) .

{The second affix of each of 'constant' and 'unsigned-constant' specifies its type and, if appropriate, its value. Only scalar constants, with the exception of real constants, have values associated with them for syntactic purposes, as only such constants can meaningfully be employed as case-labels (p35.1, p35.2) or as the bounds of a subrange type (p19.2).}

..

{CONSTANTS - continued}

- (p72.1) reverse-sign(,) : .
- (p72.2) reverse-sign(1 N, minus 1 N) : .
- (p72.3) reverse-sign(minus 1 N, 1 N) : .

{DENOTATIONS}

- (p73.1) character(VALUE) : " string-item(VALUE) " .
- (p74.1) string(N) : " string-item-sequence(N) " .
- (p75.1) string-item-sequence(1 1) : string-item(VALUE1) string-item(VALUE2) .
- (p75.2) string-item-sequence(1 N) : string-item-sequence(N) string-item(VALUE) .
- (p76.1) string-item(QUOTE) : " " .
- (p76.2) string-item(A) : a .
- (p76.27) string-item(Z) : z .

Figure A.1(d) (continued)

{ DENOTATIONS - continued }

{Here QUOTE, A, ..., Z stand for the terminal productions of 'VALUE' which represent the (implementation-dependant) internal values of the characters quote ("), a, ..., z respectively. Additional meta-rules, like p76.2 - p76.27, may be added for other characters in the implementation's character set.}

(p77.1) unsigned-integer(N) : digit(N) ;

(p77.2) unsigned-integer(N1) multiply(N1, 1 1 1 1 1 1 1 1, N2)
digit(N3) add(N2, N3, N) .

(p78.1) unsigned-real : unsigned-integer(N) scale-factor ;

(p78.2) unsigned-integer(N) fraction scale-factor ;

(p78.3) unsigned-integer(N) fraction .

(p79.1) fraction : . digit(N) ;

(p79.2) fraction digit(N) .

(p80.1) scale-factor : e unsigned-integer(N) ;

(p80.2) e sign unsigned-integer(N) .

Figure A.1(d) (continued)

{DENOTATIONS - continued}

(p81.1) digit() : 0 .
 (p81.2) digit(1) : 1 .
 (p81.10) digit(1 1 1 1 1 1 1 1) : 9 .
 (p82.1) add(N, , N) : .
 (p82.2) add(N1, 1 N2, N) : add(1 N1, N2, N) .
 (p83.1) multiply(N, ,) : .
 (p83.2) multiply(N1, 1 N2, N) : multiply(N1, N2, N3) add(N3, N1, N) .

Figure A.1(d) (concluded)

- Knuth 71a Knuth, D.E.:
"Examples of Formal Semantics"
in "Symposium on Semantics of Algorithmic
Languages",
Springer-Verlag, 1971.
- Knuth 71b Knuth, D.E.:
"Top-Down Syntax Analysis",
Acta Informatica 1 (1971), 79-110.
- Naur 60 Naur, P. (Ed.):
"Report on the Algorithmic Language ALGOL 60",
Comm. ACM 3, 5 (May 1960), 299-314.
- Naur 63 Naur, P. (Ed.):
"Revised Report on the Algorithmic Language
ALGOL 60",
Comm. ACM 6, 1 (January 1963), 1-17.
- Sintzoff 67 Sintzoff, M.:
"Existence of a van Wijngaarden Syntax for
every Recursively Enumerable Set",
Annales Soc. Scientifique de Bruxelles 81, 2
(1967), 115-118.
- Stearns 67 Stearns, R.E., and Lewis, P.M.:
"Property Grammars and Table Machines",
Inf. Contr. 14 (1969), 524-549.
- van Wijngaarden 68 van Wijngaarden, A. (Ed.), Mailloux, B.,
Peck, J.E.L., and Koster, C.H.A.:
"Report on the Algorithmic Language ALGOL 68",
MR101, Mathematisch Centrum, Amsterdam, 1969.
- Wirth 66 Wirth, N., and Hoare, C.A.R.:
"A Contribution to the Development of ALGOL",
Comm. ACM 9, 6 (June 1966), 413-431.
- Wirth 71 Wirth, N.:
"The Programming Language PASCAL",
Acta Informatica 1 (1971), 35-63.
- Wirth 73 Wirth, N.:
"The Programming Language PASCAL" (Revised
Report),
Eidgenössische Technische Hochschule,
Zürich, July 1973.

INDEX OF DEFINITIONS

- #-symbols 55
- A-LR(k) auxiliary grammar 71
- AF-LALR(k) auxiliary grammar 59
- AF-LR(0) parsing algorithm 64
- AF-LR(k) auxiliary grammar 59
- AF-LR(k) parsing algorithm 67
- AF-SLR(k) auxiliary grammar 59
- AFSM 64
- AG 21
- affix (AG) 21
- affix (EAG) 79
- affix grammar 21
- affix-form (EAG) 80
- affix-nonterminal (AG) 21
- affix-nonterminal (EAG) 79
- affix-rule (AG) 21
- affix-rule (EAG) 79
- affix-terminal (AG) 21
- affix-terminal (EAG) 79
- affix-variable (AG) 21
- affix-variable (EAG) 79
- analyse-predicate 86
- applied occurrence (of affix-variable) (AG) 26
- applied occurrence (of affix-variable) (EAG) 81
- associated affix-nonterminal (AG) 21
- associated function 22
- auxiliary grammar 45
- blind alley (AG) 23
- blind alley (EAG) 80
- canonical derivation (AG) 24
- canonical direct derivation (AG) 24
- canonical form (AG) 24
- canonical parse (AG) 24
- CFSM 37
- characteristic grammar 37
- characteristic head string 56
- characteristic string 37
- control (AG) 22
- control (EAG) 79
- copy-meta-rule 46
- copy-rule 47
- copy-state 63
- copy-symbol 45
- copy-transition 63
- defining occurrence (of affix-variable) (AG) 26
- defining occurrence (of affix-variable) (EAG) 81
- derivation (AG) 24
- derivation (auxiliary grammar) 47
- derived affix-position 22
- direct derivation (AG) 24
- direct derivation (auxiliary grammar) 47
- direct production (AG) 23
- direct production (EAG) 80
- disjoint primitive predicate symbols 72
- distinguished nonterminal (AG) 21

- distinguished nonterminal (EAG) 79
- domain (of variable) (AG) 21
- domain (of affix-position) (AG) 22
- domain (of affix-position) (EAG) 79
- EAG 79
- equal-predicate 90
- extended affix grammar 79
- head (of hypernotation) 22
- head grammar 58
- head string 46
- hypernotation (AG) 22
- hypernotation (EAG) 80
- inadequate state (of CFSM) 37
- inherited affix-position 22
- k-look-ahead state 40
- k-symbol look-ahead set 39
- LALR(k) grammar 39
- LALR(k) parsing algorithm 40
- LRkCFSM 41
- LR(0) parsing algorithm 38
- LR(k) grammar 36, 37
- LR(k) parsing algorithm 41
- language (AG) 23
- language (auxiliary grammar) 47
- language (EAG) 80
- left side (of meta-rule) (AG) 22
- left side (of meta-rule) (EAG) 80
- left-side head (of meta-rule) 46
- look-ahead state 39
- meta-rule (AG) 22
- meta-rule (auxiliary grammar) 45
- meta-rule (EAG) 80
- multi-predicate state 71
- nonterminal (AG) 21
- nonterminal (EAG) 79
- notion (AG) 23
- notion (EAG) 80
- parse (AG) 24
- predicate state 63
- predicate-transition 63
- primitive predicate symbol (AG) 21
- production (AG) 23
- production (EAG) 80
- production rule (AG) 23
- production rule (auxiliary grammar) 46
- production rule (EAG) 80
- protonotation (AG) 23
- protonotation (EAG) 80
- read state (of CFSM) 37
- reduce state (of CFSM) 37
- right side (of meta-rule) (AG) 22
- right side (of meta-rule) (EAG) 80
- sentence (AG) 23
- sentence (EAG) 80
- SLR(k) 39
- simple look-ahead set 39
- synthesise-predicate 86
- tail string 46
- terminal (AG) 21
- terminal (EAG) 79
- terminal production (AG) 23
- terminal production (EAG) 80
- unambiguous (AG) 24
- variable (AG) 21
- variable (EAG) 79
- well-formed AG 27
- well-formed EAG 81